UiO : Department of Informatics
University of Oslo

Protein Alignment on the Intel Xeon Phi Coprocessor

Jorun Ramstad

Master's Thesis Autumn 2015

Protein Alignment on the Intel Xeon Phi Coprocessor

Jorun Ramstad

August 3, 2015

Abstract

There is an increasing need for sensitive, high perfomance sequence alignemnet tools. With the growing databases of scientificly analyzed protein sequences, more compute power is necessary. Specialized architectures arise, and a transition from serial to specialized implementationsis is required.

This thesis is a study of whether Intel 60's cores Xeon Phi coprocessor is a suitable architecture for implementation of a sequence alignment tool. The performance relative to existing tools are evaluated, as well as measurements comparing the implementation to the theoretical peak performance of the architecture.

SWIMIC, a sequence alignment tool utilizing the Smith-Waterman algorithm implemented for Intel's MIC (Many Integrated Core) architecture was made. It runs natively on a Xeon Phi coprocessor and is optimized with SIMD intrinsics, threading with OpenMP and pragma directives for vectorization.

With potential memory and compute power unexploited, SWIMIC achieves 43 GCUPS, 74 % of a similar tool also running on the Xeon Phi, and 40 % of the leading tool running on CPU's.

The study shows that the Xeon Phi coprocessor is not a suitable architecture to perform sequence alignments on, while utilizing the Smith-Waterman algorithm, due to relatively high memory footprint. The shared memory architecture possess a relatively small combined cache and with the lack of support for smaller data types this is a limitation that the four hardware thread and a 512 bit vector unit per core can not overcome.

Acknowledgements

First and foremost, I would like to thank my partner in crime for the last two years, Reidar André Brenna. Our collaboration throughout this thesis and 30 credits of course work as a pair, have led to many hours of discussions and frustration, but most of all a memorable journey.

My supervisor, Torbjørn Rognes and Jon Hjelmervik, deserves a thanks, as do all the other master students in the 10th floor for sharing their experiences and motivating me throughout this thesis. Without your company day after day in the 10th floor this journey would have been a lonely affair.

I would also like to thank all my friends and family, especially Realistforeningen for being my second home and family for the last 5 years.

A thank you to Ole Widar Saastad at USIT for the support regarding accsess to the Abel Clusters Xeon Phi coprocessors to performed this thesis calculations. You saved me from a lot of frustration.

Jorun Ramstad
University of Oslo
August 2015

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Comparing protein sequences using local sequence alignment is an important and time-consuming task in bioinformatics. It is the first step towards structural and functional analysis of newly determined sequences of amino acids. Comparing a query sequence against a large database reveals sequences with regions of similarity that indicates a relationship through a common ancestor, known as homologs. Usually homologs share a set of biological properties, and the information known about one sequence can therefore be transferred to others.

The two most common algorithms used to solve this task are the Basic Local Alignment Search Tool (BLAST) by Altschul et al. [1] and the Smith and Waterman (SW) algorithm [2]. The first uses a heuristic approach, while the second uses the principles of dynamic programming. A number of different tools exist utilizing either one of the algorithms. Most of these tools are created for CPU's and can be run on a variety of standard computers. Due to time consumption, parallelization and optimal use of resources are a high priority. Several new tools have been published in the last decade, some of which function well on CPU's. Other tools are made for specialized hardware to achieve an even higher performance, for instance applications made for the Graphics Processing Unit (GPU).

In January 2013, Intel released the Xeon Phi coprocessor as a serious contender in the high performance programming field. It is designed to tackle highly parallel problems, and is a promising hardware to implement and optimize a sequence alignment tool on. Despite the prospects of the Xeon Phi, the challenge remains to find optimal optimization technique that utilizes all aspects of the coprocessor. Some techniques for regular CPUs or GPUs with or without modifications, could prove to be suitable for the Xeon Phi coprocessor. Examination of complete utilization of the Xeon Phi coprocessor is required, in addition to research how previously

implemented alignment tools using both BLAST and Smith-Waterman utilize parallelization.

1.2 Goal

This project examines existing methods and tools used for protein similarity searches, and implements a new tool using the Intel Xeon Phi coprocessor. The overall aim of the study is to compare the performance of the implemented tool relative to exciting methods, in addition to examining measurements comparing the implementation to the theoretical peak performance of the architecture. This includes reaching the following:

- Implement and optimize a sequence alignment tool for the Intel Xeon Phi coprocessor.
- Examine whether the Xeon Phi coprocessor is a suitable hardware for sequence alignment.
- Determine how well the implemented tool utilizes the unique coprocessor architecture.
- Determine whether or not the finished tool is competitive compared to other sequence alignment tools.

1.3 Tools for Comparison

Recently Liu and Smith [3] released a new tool, SWAPHI, that utilize a Smith-Waterman approach implemented on an Intel Xeon Phi coprocessor. This tool is of interest because of how they utilize the Xeon Phi's unique design, and is a good implementation to compare this thesis future alignment tool against. Even though a Xeon Phi implementation is the closest thing to compare this thesis application against, it is also interesting to know the competitiveness against the leading tools for other hardware such as BLAST[1] and Rognes'[4] tool SWIPE for CPU's and the latest version of CUDASW++ by Liu et al. [5] designed for GPU's.

1.4 Overview of Thesis

Chapter 2 presents the **background** material including both sequence alignment and hardware specific information about the Xeon Phi.

Chapter 3 discusses the **methods** used to solve sequence alignment and some common optimization techniques useful for this thesis.

Chapter 4 presents the **implementation** of this thesis alignment tool running on a Xeon Phi coprocessor.

Chapter 5 presents and discusses the **results**, including some reflecting on the applications validity and competitiveness.

Chapter 6 presents the **conclusion** of the thesis.

Chapter 7 presents ideas for **future work**.

Chapter 2

Background

In this chapter the two fundamental aspects of creating a sequence alignment tool for the Intel Xeon Phi coprocessor are presented. First, how to align protein sequences and calculate an appropriate similarity score and second, how the Xeon Phi's unique architecture is structured and best utilized.

2.1 Sequence Alignment

An important process in searching for related protein sequences is to compare them using sequence alignment, in which sequences are compared by searching for common character patterns and establishing residue-residue correspondence among related sequences. This pairwise sequence alignment is the process of aligning two sequences and is the basis of database similarity searching.

2.1.1 Alignment Types

To find related sequences either a global or local alignment may be used to identify regions of similarity within a long sequence. A pairwise perfect match of amino acids, is to be preferred. However insertion or deletion of entries are sometimes inevitable. What distinguishes local alignment from global alignment is which parts of both the query and the database sequence is included in the result.

Given the two sequences:

```
F T F T A L I L L A V A V
F T A L L L A A V
```

The global alignment contains both sequences with the first and last letter mapped together and the rest aligned to the optimal match.

F T F T A L I L L A V A V
F - - T A L - L L A - A V

For local alignment the subsequences, i.e. the part of the sequences that gives the best match, is returned. In this case:

F T A L I L L - A V
F T A L - L L A A V

When comparing protein sequences local alignment is used to find subsequences in the database which are similar to subsequences in the query. The reason local alignment is preferred over global is that the query may differ significantly in length compared to the majority of the database sequences. With local alignment only regions that are highly similar are taken into account and unequal parts are discarded.

Substitution Scoring Matrix

The alignment procedure has to make use of a scoring system, which is a set of values for quantifying the likelihood of one residue being substituted by another in an alignment. These substitution scoring matrices are derived from statistical analysis and describes the probability rate of which an amino acid a in a sequence is changed to another amino acid b in a certain evolutionary time.

A positive score indicates a more likely frequency of substitution than what would have occurred in nature by random chance. A score of zero refers to a substitution equal to what is expected by chance. A negative score means a frequency of substitution less likely to have occurred by random chance and is normally the case between dissimilar residues.

The two most widely used series of substitution scoring matrices for amino acids are PAM and BLOSUM. Figure 2.1 [1] shows an example, the BLOSUM62 matrix. Both series exemplifies the main aspects required for a well functioning scoring matrix. The first being the degree of the "biological relationship" between the amino acids, and the second being the probability of two amino acids occurring at homologous positions in sequences that have a common ancestor, or that one is the ancestor of the other[6].

Gap Penalty

In order to acquire the best possible match one is often required to either insert or delete an amino acid entry to align the query sequence with a database entry. This insertions and deletions requires consideration when calculating the score of similarity, and is referred to as gap penalty. The

[1] http://en.wikipedia.org/wiki/File:BLOSUM62.gif, 2008-03-07

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4																			
Arg	-1	5																		
Asn	-2	0	6																	
Asp	-2	-2	1	6																
Cys	0	-3	-3	-3	9															
Gln	-1	1	0	0	-3	5														
Glu	-1	0	0	2	-4	2	5													
Gly	0	-2	0	-1	-3	-2	-2	6												
His	-2	0	1	-1	-3	0	0	-2	8											
Ile	-1	-3	-3	-3	-1	-3	-3	-4	-3	4										
Leu	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4									
Lys	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5								
Met	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5							
Phe	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6						
Pro	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7					
Ser	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4				
Thr	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5			
Trp	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11		
Tyr	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	
Val	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

Figure 2.1: The BLOSUM62 substitution scoring matrix.

gap penalty is commonly assigned by an affine function that give an initial penalty for a gap opening, and an additional penalty which increases in correspondence with the length of the gap. A typical penalty used as default in SWIPE [4] is *11* for an opening and penalty of *1* for each extension.

2.1.2 Databases

The sequence databases used to identify new proteins are large and contains millions of sequences of an average length of 300-400 amino acids. A commonly used protein sequence database is UniProtKB/Swiss-Prot [7]. It is a manually annotated and non-redundant database. The aim of UniProtKB/Swiss-Prot is to provide all known relevant information about a particular protein. The manual annotation of an entry involves detailed analysis of the protein sequence and the scientific literature. Annotations are regularly reviewed to keep up with current scientific findings. From the release 2014_05 of 14-May-14 [2] the UniProtKB/Swiss-Prot database contains 545388 sequence entries, comprising 193948795 amino acids abstracted from 228536 references.

Another possible database for proteins is *GenBank* [8]. In 2013 it contained over 150 billion nucleotide bases in more than 162 million sequences. It is produced and maintained by the National Center for Biotechnology Information (NCBI) as part of the International Nucleotide Sequence Database Collaboration (INSDC). Also built by NCBI is *The*

[2] http://web.expasy.org/docs/relnotes/relstat.html

Reference Sequence (*RefSeq*) database [9]. It differs from *GenBank* in that it only provides a single record for each natural biological molecule, i.e. DNA, RNA or protein.

2.1.3 Input Format

The databases contains sequences in FASTA format that originates from the alignment tool FASTP [10]. FASTA is a text based format in which the amino acids are represented by single letter codes. Each sequence begins with a single line description followed by the sequence data, see Figure 2.2 for an example. The description line is distinguished from the sequence data by an initial '>' symbol. The identifier of the sequence is the word directly following the '>' and the rest of the line are an optional description. The end of the sequence is recognized by the start of the next sequence beginning with '>'.

```
>MCHU - Calmodulin - Human, rabbit, bovine, rat, and chicken
ADQLTEEQIAEFKEAFSLFDKDGDGTITTKELGTVMRSLGQN
PTEAELQDMINEVDADGNGTIDFPEFLTMMARKMKDTDSEE
EIREAFRVFDKDGNGYISAAELRHVMTNLGEKLTDEEVDEMI
READIDGDGQVNYEEFVQMMTAK
```

Figure 2.2: An example sequence in FASTA format [3]

2.2 The Intel Xeon Phi Coprocessor

The hardware utilized in this thesis is the Intel Xeon Phi coprocessor. It has attracted attention in the super computing world and several of the most powerful supercomputers utilizes the Xeon Phi as their computation unit [4].

2.2.1 Technical specifications

A few key technical specifications about the Xeon Phi, from the Best Practice Guide Intel Xeon Phi v1.1 by Barth et al. [11], supplemented by Jeffers' and Reinders' [12] book, is described below.

[3] http://en.wikipedia.org/wiki/FASTA_format, 2014-05-09
[4] http://www.top500.org/lists/2014/11/

General

The Intel Xeon Phi coprocessor can be looked at from a programmers point of view as an x86-based SMP-on-a-chip with roughly 60 cores, with multiple hardware threads per core, and 512-bit SIMD instructions. All 60 cores has the same fundamentals as the original Pentium design and are in-order dual issued x86 processor cores which means it can sustain executing two instructions per cycle. In addition from the Pentium design it also offers a 64-bit support, four hardware threads per core, power management, ring interconnect support and 512-bit SIMD capabilities. An overview is shown in Figure 2.3. Each core is connected in a symmetric multiprocessing (SMP) fashion, which involves an architecture where two or more identical processors are connected to a single, shared main memory, have full access to all I/O devices, and are controlled by a single operating system instance that treats all processors equally, reserving none for special purposes. To connect them all a high performance on-die bidirectional ring interconnect is used, the Core Ring Interface (CRI).

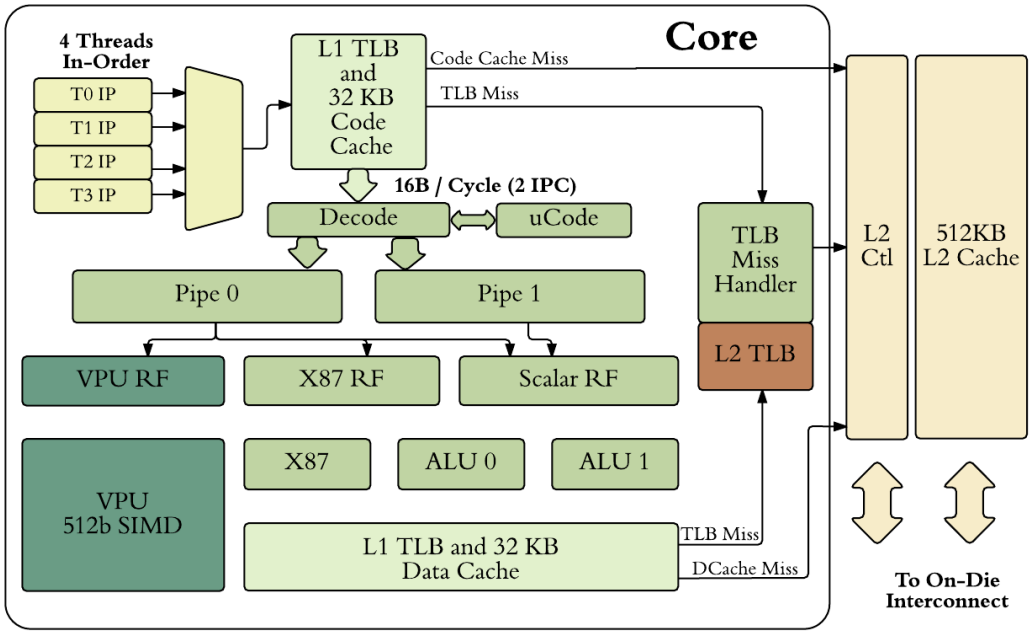

Figure 2.3: Architecture of a single Xeon Phi core drawn from the figures in the Jeffers' and Reinders' book[12] page 8.

The Xeon Phi coprocessor needs to be connected to an Intel Xeon processor-based host platform, which is done through a high-speed, point-to-point communication channel, a PCI Express bus. This gives the opportunity to either run a program from the host or natively on the coprocessor.

The coprocessor runs a full service Linux operating system designed for a Many Integrated Core (MIC) architecture and it is supported by standard Intel development tools including Intel Parallel Studio XE, C/C++ compilers and OpenMP that all may be highly useful when crating an optimized tool for alignment search in protein databases.

Vector Unit

The most interesting new feature the Xeon Phi posses is the new 512-bit wide Vector Processing Unit (VPU) that looks promising with a possible use of SIMD instructions to vectorize the Smith-Waterman algorithm. Previous Intel SIMD extensions are not supported, but a new instructions set including gather/scatter, fused multiply-add, masked vector instruction etc. are supported. With the SIMD width of 64-Byte (512-Bit) all data needs to be aligned to 64-Byte to achieve a good performance. Most vector instructions on the Xeon Phi has a 4-clock latency with a 1 clock throughput.

The coprocessor also has a lot of built-in auto-vectorization features. Pragmas like *#pragma vector aligned* or *#pragma simd* may be used to accomplish this. Auto-vectorization is enabled at default optimization level *-O2*. Each core's VPU also includes the Extended Math Unit (EMU) that makes it possible to executes 16 32-Bit integer operations or 8 double-precision floating point operations per cycle.

Cache

The Xeon Phis cache hierarchy consist of the L1 cache that each core utilizes solely, and a shared L2 cache and tag directory for all the cores. The L1 cache consist of a 32 KB L1 instruction cache and 32 KB L1 data cache. It has a load-to-use latency of 1 cycle, which means that an integer value loaded from the L1 cache can be used in the next clock cycle by an integer instruction (vector instructions have different latencies than integer instructions). The L2 cache contributes 512 KB to the global shared L2 cache storage, inclusive of the L1 data and instruction cache. The effective total L2 size of the chip is only 512 KB if every core shares exactly the same code and data in perfect synchronization, if no cores share any data or code the effective total L2 size of the chip is up to 31 MB. The actual size of the workload-perceived L2 storage is a function of the degree of code and data sharing among cores and threads. The raw latency for the L2 cache is 11 clock cycles.

Memory

When it comes to memory access, the Xeon Phi has a 8 GB capacity and a memory channel interface speed of 5.5 gigatransfers per second (GT/s) on a 60 cores coprocessor. There are 8 memory controllers each accessing two

memory channels. Each memory transaction is 4 byte of data, resulting in 5.5 GT/s times 4 bytes or 22 GB/s per each 16 channels, giving a maximum transfer rate of 352 GB/s. An effective peak of 50 to 60 percent is realistic to expect. The main memory is interleaved across the cores and accessed through the ring interface as well, with hook memory controllers on the die. By linking memory ports onto the ring, the interleaving around the cores and ring smooths out the operation of the coprocessor when all the cores are working.

Architecture

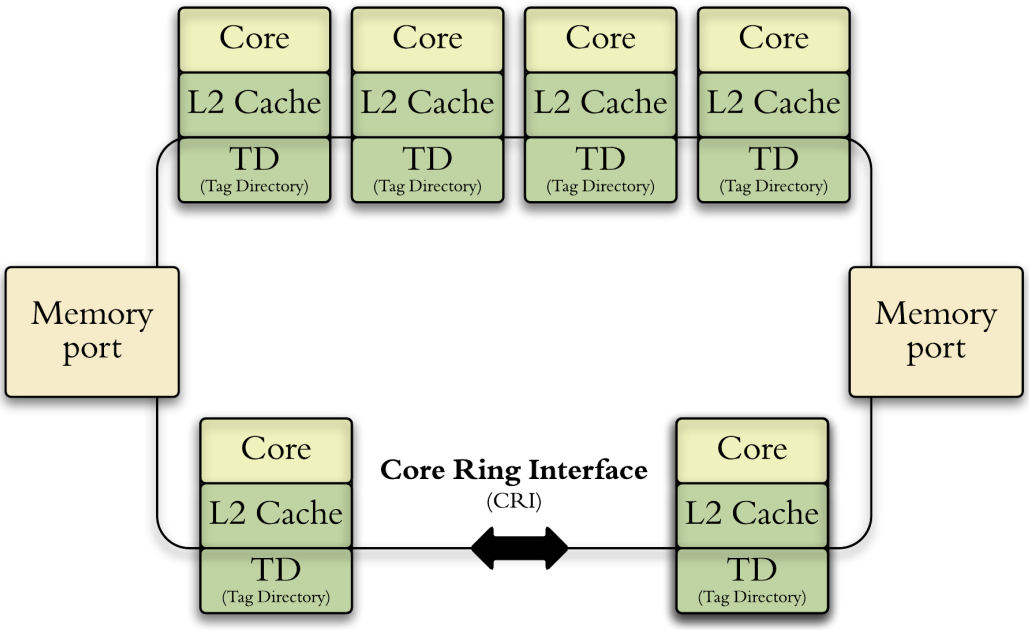

Figure 2.4: Simplified overview of the Xeon Phi architecture drawn from the figures in the Jeffers' and Reinders' book[12] page 9.

The sketch in Figure 2.4 shows a simplified Xeon Phi architecture with only 6 cores. The connection via the on-die interconnect ring interface (CRI) contains the shared L2 cache and Tag Directory (TD) together with memory ports. The cores, illustrated in Figure 2.3, are complex and some of the components included are the four hardware threads, the vector processing unit and the L1 cache.

2.2.2 Compilation

When implementing an application for the Xeon Phi coprocessor there are two models available, either a *native execution model* or an *offloading model*. A native execution model is where the application is meant to run natively on the coprocessor alone, where as with the offload model the program may be viewed as running on processor(s) and selected work is offloaded to the co-processor(s). Some aspects with both solutions are discussed below and in Figure 2.5.

Offload (Heterogeneous)	Native (Autonomous)
• Better serial processing • More memory • Better file access • Makes fuller use of available resources	• Simpler programming model • Easier or no code porting • More constraints • No transfer latency

Figure 2.5: Offload vs native programming models

The native model may be appropriate if the application contains very little serial processing and has a modest memory footprint. Since it is slower to do I/O work on the Xeon Phi, the application should not do extensive I/O work and Graphical user interface (GUI). The great advantage with the native model is that there is no need to transfer data between the CPU and the coprocessor, hence no transfer latency. It is also good for applications that are largely doing operations that map to parallelism either in threads or vectors. On the other hand, the native execution model is more constraining and the memory available is very limited.

If the application is more extensive and utilizes more resources an offload model is definitely the best choice. Unfortunately offload has some additional concerns when it comes to allocation, copies and deallocation of data and that it requires two levels of memory blocking: one to fit the input data onto the coprocessor, and another within the offload code to fit within the processor caches and not oversaturate the processor memory subsystem when all cores are busy.

For communication either OpenMP (Open Multi-Processing) or MPI (Message Passing Interface) may be used to transfer data between the host and the coprocessor.

The MPI communication protocol is used to program parallel applications

and the standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface [5]. MPI is widely used on distributed memory system and computers such as computer clusters.

OpenMP on the other hand is an API that supports multi-platform shared memory multiprocessing programming on most processor architectures and operating systems. It is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs [6].

2.2.3 Gaining Optimal Performance

When programming for the Intel Xeon Phi coprocessor there are some important aspects to consider concerning the architecture including optimization techniques, scaling, alignment of data and memory usage.

A single Xeon Phi core is slower than a Xeon core due to lower clock frequency, smaller caches and lack of sophisticated features such as out-of-order execution and branch prediction. To fully exploit the processing power of a Xeon Phi parallelism on both instruction level (SIMD) and thread level (OpenMP) is needed. Xeon Phi can only perform memory reads/writes on 64-byte aligned data therefore any unaligned data will be fetched and stored by performing a masked unpack or pack operation on the first and last unaligned bytes. This may cause performance degradation, especially if the data to be operated on is small in size and mostly unaligned.

Because of the relatively small cache on the Xeon Phi it is important to be aware of data order in memory. Close by data is read into cache for later use, hence the importance of conscious memory usage. A good question when optimizing is: *"Is the data in cache the next data to be used for calculations or is it something completely different?"*. If the cached data is rarely used unnecessary memory reads drastically slows down the execution time.

Since the vector unit is capable of performing 16 single precision floats or 8 double precision floats per clock cycle, vectorization of an application can give as much as 8 or 16 times speedup. The VPU also posses the ability of *Fused Multiply-Add* or *Fused Multiply-Subtract* operations which effectively double the theoretical floating point performance. To reach the potential speedup gained by utilizing vectorization is not realistic, but a significantly performance gain is definitely to be expected.

To utilize the four hardware threads the Xeon Phi possesses, the problem needs to scale well with hundreds of threads. In theory each device has more

[5] http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
[6] http://openmp.org/openmp-faq.html#OMPAPI.General

than 200 threads available and they are used to hide latency implicit in the in-order micro-architecture. In practice, use of at least two threads per core is nearly always beneficial.

Problem criteria

Scaling Are the workload divided in a fashion that it may be distributed to at least 200 threads?

Vectorization Are the application making strong use of vectorization?

Memory usage Is the memory well addressed for a shared architecture and are all data 64-byte aligned?

Cache usage Is the next data in memory the next needed data?

With all these criteria considered a performance boost from the Xeon Phi should be gained. However, if the outcome still do not meet the anticipated performance there might be some fine tuning that may improve the run time. By either taking control with intrinsics or give control in terms of extensive use of `pragma` directives, an additional performance boost may appear. Despite an unique architecture designed for parallelization, there are unfortunately not all algorithms/problems that are suitable to be executed on the Xeon Phi and other hardware might yield a better result.

Chapter 3

Methods

This chapter presents the techniques needed to create an optimized alignment tool. The two most common algorithms for sequence alignment is discussed as well as the tools that utilize them. In addition a general idea of optimization techniques are presented.

3.1 Common Approaches for Alignment

A look into the two most common methods of doing a sequence alignment search, BLAST [1] and Smith-Waterman [2], is described below with their reliability and speed as key aspects.

3.1.1 BLAST

BLAST uses a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution, called heuristic. This is achieved by trading optimality, completeness, accuracy, or precision for speed. In a way, it can be considered a shortcut. It does not guarantee the optimal alignment, but with a heuristic that approximates the Smith-Waterman algorithm the result is more that acceptable. It does not compare either sequences in its entirety, but rater locates short matches between the two. The main idea of BLAST is that statistically significant alignments often contains segment pairs that do not increase its score while either extending or shortening down its length, also known as high-scoring segment pairs (HSP).

The first step of BLAST is to find matching segment pairs, word pairs, with length w, that score at least T. The words to compare with the database sequences are found by matching all possible w letter words out of the 20 amino acids to the query sequence, and save the words that have a score higher than T for at least one word in the query sequence. For protein sequences the word length, w, is usually 3. The database is then searched

for occurrences of the saved words from the query sequence to find hits. The hits is extended to high-scoring segment pairs to check if they score higher than a threshold V. This threshold is determined such that there is reason to believe homology [6].

Depending on T and w, the sensitivity is determined. While increasing T the runtime of the search decreases, since fewer word pairs is found and extended. This will also decrease sensitivity as word pairs might be overlooked. The last step is to use dynamic programming to align the HSPs that score more than the given threshold to introduce gaps.

The original BLAST only generates ungapped alignments individually, including the initially found HSPs, even when more than one HSP is found in a database sequence. Later versions of BLAST [13] produces a single alignment with gaps that can include all of the initially found HSP regions.

3.1.2 Smith-Waterman

Dynamic programming can be used to find optimal alignment, both global and local, with few changes to the algorithm. Needleman and Wunsch [14] were the first to use dynamic programming in bioinformatics to find optimal global alignment. Their algorithm provided the foundation for the first approach for optimal local alignment, done by Smith and Waterman in 1981, hence the later well known Smith-Waterman algorithm[2]. The complexity of the algorithm where $O(m^2n)$, which where later improved by Gotoh [15] to run at $O(mn)$ by just testing if a gap is elongated, and thus increase the penalty if true, instead of testing for all possible gap lengths. The modification is described in more detail in the next section.

Dynamic programming is often used to solve optimization problems where the problem may have a number of feasible solutions, and the desired solution is the *best* one. Dynamic programming is a very powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one. Starting with the smallest first and using the answers to smaller problems to help figure out the larger ones, until the whole set of problems are solved.

In the Smith-Waterman algorithm the results is stored in a matrix and the score found for a cell earlier in the solution process is used in later calculations.

The Smith-Waterman algorithm is divided into two parts. First identify the highest possible score using dynamic programming, utilizing both a scoring matrix s, like BLOSUM62, and affine gap penalty W. Both presented in Section 2.1.1. If the score for a matrix cell is negative, the cell score is set to zero.

$$H(i,0) = 0,\ 0 \leq i \leq m.$$

$$H(0,j) = 0,\ 0 \le j \le n.$$

$$H(i,j) = \max \begin{Bmatrix} 0 & \\ H(i-1,j-1) + s(a_i,b_j) & \text{Match/Mismatch} \\ \max_{k\ge 1}\{H(i-k,j) + W_k\} & \text{Deletion} \\ \max_{l\ge 1}\{H(i,j-l) + W_l\} & \text{Insertion} \end{Bmatrix},$$

where $1 \le i \le m$ and $1 \le j \le n$.

Example 3.1.1. Smith-Waterman example from wikipedia [1] with the two sequences,

Database sequence:	A	C	A	C	A	C	T	A
Query sequence:	A	G	C	A	C	A	C	A

calculating with a simplified scoring matrix $s(a,b) = +2$ if $a = b$ (match), -1 if $a \neq b$ (mismatch), gives the resulting matrix

$$H = \begin{pmatrix} & - & A & C & A & C & A & C & T & A \\ - & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & \mathbf{2} & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & \mathbf{3} & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & \mathbf{5} & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & \mathbf{7} & 6 & 7 & 6 & 5 \\ A & 0 & 2 & 3 & 6 & 6 & \mathbf{9} & 8 & 7 & 8 \\ C & 0 & 1 & 4 & 5 & 8 & 8 & \mathbf{11} & \mathbf{10} & 9 \\ A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & \mathbf{12} \end{pmatrix}.$$

Then the alignment(s) is identified by going backward from the highest score following the highest entry upward in the matrix until a 0 is the only next entry. A matrix with arrows is shown below to better illustrate the alignment.

$$T = \begin{pmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ G & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow & \uparrow \\ C & 0 & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow \end{pmatrix}.$$

[1] http://en.wikipedia.org/wiki/Smith-Waterman_algorithm, 2014-04-28

This gives the optimal alignment following the arrows.

Database sequence:	A	-	C	A	C	A	C	T	A
Query sequence:	A	G	C	A	C	A	C	-	A

A diagonal arrow reflect a match between the query and the database sequence, a upward arrow implies a deletion and a left arrow implies an insertion.

Gotoh's Modification of the Smith-Waterman Algorithm

The modification of Gotoh [15] for affine gap penalty functions are shown below.

$$H_{i,j} = \begin{cases} \max \begin{cases} H_{i-1,j-1} + SM[q_i, d_j] \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} & \begin{array}{l} i > 0 \\ \cap \\ j > 0 \end{array} \\ 0 & \begin{array}{l} i = 0 \\ \cup \\ j = 0 \end{array} \end{cases}$$

$$E_{i,j} = \begin{cases} \max \begin{cases} H_{i,j-1} - Q \\ E_{i,j-1} - R \end{cases} & j > 0 \\ 0 & j = 0 \end{cases}$$

$$F_{i,j} = \begin{cases} \max \begin{cases} H_{i-1,j} - Q \\ F_{i-1,j} - R \end{cases} & i > 0 \\ 0 & i = 0 \end{cases}$$

$$S = \max_{1 \le i \le m \cap 1 \le j \le n} H_{i,j}$$

The major difference from the original Smith-Waterman algorithm is how the gap penalty is calculated. By adding $E_{i,j}$ and $F_{i,j}$ matrices, which holds the score of aligning the same prefixes of the query sequence q and the database sequence d but ending with a gap in the query and the database, respectively, the penalty can be updated by only increasing the value of $E_{i,j}$ or $F_{i,j}$ instead of testing for all possible gap lengths. Q equals a gap opening and one extension, while R is only the gap extension penalty. SM in the equation is the score from the substitution scoring matrix of aligning q_i with d_j, while S is the overall optimal alignment score and equals the highest value in $H_{i,j}$.

3.2 Existing Tools

The most relevant tool to compare this thesis tool to is SWAPHI [3] since it is also implemented on the Xeon Phi coprocessor. It is an application made to use the offload model where all structural work is done on the Xeon host processor, while the Smith-Waterman calculations is offloaded to the Xeon Phi coprocessor. This offers the opportunity to connect multiple Xeon Phi coprocessor together and distribute the work to more coprocessors.

A Smith-Waterman tool, which is one of the fastest on the marked for CPUs, is Rognes' tool SWIPE [4]. The algorithm is implemented on Intel processors with SSSE3 with parallelization over multiple database sequences. Instead of aligning one database sequence against the query sequence at a time, residues from multiple database sequences are retrieved and processed in parallel. Rapid extraction and organization of data from the database sequences have made this approach feasible. The approach also involves computing four consecutive cells along the database sequences before proceeding to the next query residue in order to reduce the number of memory accesses needed.

A second tool that utilizes Smith-Waterman with good results is CUDASW++ 3.0 [5]. It is coupling CPU and GPU SIMD instructions and conducting concurrent CPU and GPU computations. To balance the runtime differences of the CPU and the GPU the distribution of sequences is done dynamically by their compute power. The optimizations done for the CPU is employed by the steaming SIMD extension (SSE)-based vector execution units and multi-threading. For the GPU a SIMD parallelization approach using PTX SIMD video instruction is used to obtain more data parallelization.

3.3 Optimization Techniques

The concept of optimizing a computer program is to modify the implementation to make it run more efficiently or use fewer resources. For instance, a computer program may be optimized so that it runs faster, requires less memory or other resources, or consumes less power. Which technique that is most suitable depends on the implemented problem and the hardware accessible. It is important to consider the space-time tradeoff. This is the tradeoff between calculation in less time by using more storage space (or memory) and solving a problem using very little storage space but with longer run time.

Parallelization

The most common optimizing technique is parallelization, in which more than one calculation is carried out simultaneously. Either by dividing a large problem into smaller separate tasks and then solve the non depending parts

simultaneously, or perform the same calculation for various conditions in parallel.

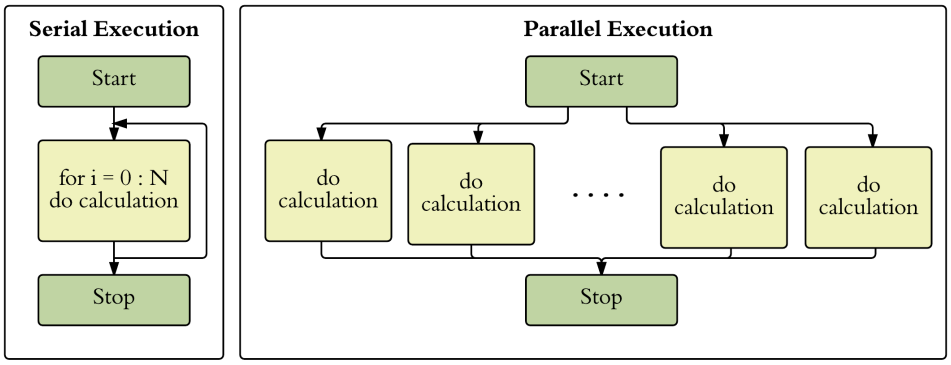

Figure 3.1: A drawing of single vs. parallel execution

Parallelization may be done on several levels, such as data level like vectorization and SMID, thread level like OpenMP and process level like MPI, all depending on the problem and the supported parallelism of the hardware.

The common hardware used for parallelism are multi-core and multi-processor computers having multiple processing elements within a single machine, and clusters that uses multiple computers to work on the same task. Specialized parallel computer architectures have been created to achieve an even higher level of parallelism, like the Intel Xeon Phi used in this thesis, where a shared memory is connected between 60 cores that all are created for performing individual work simultaneously.

The maximum possible speedup of a program computing in parallel is limited by the time needed for the sequential fraction of the program, also known as Amdahl's law [16] defined in Definition 4.3.1. From this one can see that if only 70 % of the program is possible to execute in parallel then the maximum speedup is $1/30$.

Some algorithms highly depend on previous calculations, thus making it impossible to do more calculations in parallel. The Smith-Waterman algorithm discussed in the previous section is one of those, and creative thinking is required to find a way to parallelize it. Since the alignment search is applied to a large database, a common way to parallelize the search is to compare one query sequence against a multiple of database sequences in parallel. The same goes for the other way round, comparing multiple query sequences against one database sequence at a time. This combined with vectorization is what made SWIPE [4] the leading tools utilizing the Smith-Waterman algorithm on a regular CPU.

Definition 3.3.1. Amdahl's Law
Given:
$n \in \mathbb{N}$, the number of threads of execution,
$B \in [0, 1]$, the fraction of the algorithm that is strictly serial,
The time $T(n)$ an algorithm takes to finish when being executed on n thread(s) of execution corresponds to:

$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

Therefore, the theoretical speedup $S(n)$ of executing a given algorithm on a system capable of executing n threads of execution is:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1 - B)\right)} = \frac{1}{B + \frac{1}{n}(1 - B)}$$

Vectorization

Differing from a scalar implementation, in which only one single operation is performed at once, vectorization is a form of data-parallel programming which makes the processor perform the same operation simultaneously on N data elements of a vector. A vector may be looked at as a one dimensional array of scalar objects, like the first row in Figure 3.2.

This method of Single Instruction, Multiple Data (SIMD) is used when the same operation is performed on a large amount of data. It is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio.

	8	15	6	1	9	13	7	4
	4	3	21	5	6	42	2	4
MAX	8	15	21	5	9	42	7	4

Figure 3.2: Single instruction, multiple data (SIMD)

In this thesis vectorization will be used to execute the steps in the Smith-Waterman algorithm on a multiple of database sequences. Even though the Smith-Waterman algorithm is highly depending of the previous calculations the steps in each iteration of the algorithm are pretty simple and consist of approximately 10 sub, max or add vector instructions, which all are possible

to do with Intel's SIMD intrinsics. Figure 3.2 shows an example of finding the maximum value of two integers.

Chapter 4

Implementation

The implementation process of this thesis' alignment tool featured a handful of steps. Firstly to make a prototype to get to know the flow of an alignment process and the algorithm involved, the transition to an implementation utilizing the unique Xeon Phi architecture, and lastly the optimization to gain the best possible performance.

4.1 Prototype

The first implementing task was to make a simple prototype with the goal to locate and attain the optimal local alignment without optimization and workload distribution. From the dive into the pros and cons for both BLAST and some Smith-Waterman approaches, the decision to use dynamic programming for the Xeon Phi implementation came naturally. The prototype was a way to get to know the algorithm and the challenges associated with it.

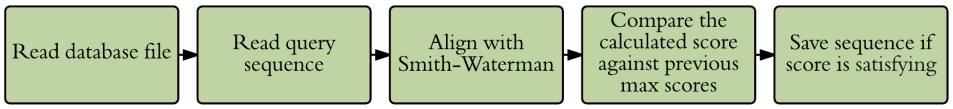

Figure 4.1: The prototypes main steps

The main functionality the prototype needed was the steps shown in Figure 4.1. A way of retrieving both query and database sequences, a function utilizing the Smith-Waterman algorithm to align the sequences and pass on the scoring value, and of course a comparison to ensure the sequence with the highest score computed got saved.

For sequence retrieval a file in FASTA format was read line by line and stored in a sequence struct containing the sequence of amino acids and

the length of the sequence. The sequences description is redundant while aligning a sequence, therefore it is skipped and never saved to memory.

To align the sequences, the Smith-Waterman algorithm together with Gotoh's [15] modification, is used. The calculation is carried out column wise as shown in Figure 4.2. Only the previously calculated row of the H and E array is necessary in the calculation, and therefore the only space allocated in memory. The highest score achieved for any element of H is stored and returned as the alignment score.

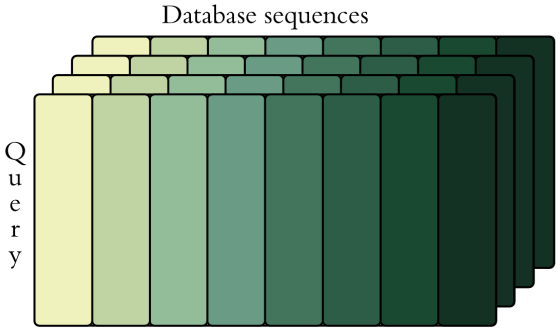

Figure 4.2: The alignment is carried out column-wise.

As an extension, an additional function doing the same as the earlier mentioned Smith-Waterman function is called for the sequence with the best score to compose the full H array to locate the local alignment within the sequence. This H array is sent to a function that locates the highest score, backtrace until a 0 is found and storing a three string representation containing the query sequence, a symbol representation of the alignment and the database sequence, all with respect to insertions and deletions.

Due to all of the existing sequence alignment tools the credibility of the prototype was easy to determine. If the prototype yielded the same optimal sequence(s) as a leading tool, like SWIPE, the prototype is likely to be correct. The further development was then to transfer the prototype to execute on the Xeon Phi. This included workload distribution and memory management.

4.2 SWIMIC

With the prototype finish, a tool actually designed for the Xeon Phi coprocessor needed to be implemented. The implementation was to run on the Many Integrated Core (MIC) architecture utilizing the Smith-Waterman algorithm, therefor a suitable name for the finish tool was SWIMIC. Short for *the Smith-Waterman algorithm Implemented for MIC architecture*.

4.3 Framework and Execution Model

Before the Xeon Phi implementation began some decisions concerning framework for parallelization and an execution model for the Xeon Phi had to be determined.

MPI

Initially the thought was to use MPI (Message Passing Interface) as the framework for parallelization. MPI is what Rognes' tool SWIPE [4] uses together with SIMD vectorization, and it seemed reasonable to use the same approach for this thesis work. The thought of using something else never appeared until executing a simple MPI implementation of the prototype on the Xeon Phi and the run time was slower than on a Xeon processor.

How could it be that a parallel implementation executed slower on an architecture that was made for parallelism? The explanation was simple and should have been spotted at a much earlier stage. The Xeon Phi has a shared memory pool, and MPI is constructed for communication between processes on a distributed memory system. The MPI prototype was executing poorer because it was unnecessary distributing memory that the kernels on the Xeon Phi already had access to.

OpenMP

The other parallelization framework available for the Xeon Phis is the OpenMP API. OpenMP is used to distribute the workload between the cores. If the host processor is included in the calculation, both OpenMP and MPI may be used for communication between the host and the coprocessor. For all other purposes OpenMP is the obvious choice.

OpenMP is an implementation of multi-threading where a master thread is executing the program and forks out a specified number of slave threads when a pre-specified parallel part is reached. It follows the general idea of parallel execution shown in Figure 3.1 on page 21. A section that is meant to run in parallel is marked accordingly by pragmas directives. For instance *#pragma omp parallel for schedule(dynamic) num_threads(THREADS)* runs the following for loop in parallel with THREADS number of threads and the workload is divided among them dynamically during runtime. After the execution of the parallel section, the threads join back into the master thread, which continues onward to the end of the program.

For OpenMP to be utilized to the best of its ability the size of the task to be done in parallel must be known before it starts the execution. For instance how many iteration in a loop before it is executed, thus for loops are preferred over while loops.

Execution Model

In Section 2.2.2 some pros and cons concerning either a native or offload execution model where presented. Since a sequence alignment tool more or less only perform the Smith-Waterman algorithm [2] which falls under the category of mapping to parallelism either in threads or vectors, the native execution model was chosen. The relatively small usage of I/O work, as well as a command line only application also favored the native execution model.

4.4 Preparation of the Database

To make sure the alignment process had the best starting point as possible, a preprocessing of the database was necessary.

4.4.1 Sorting

The preparation contained gaining the knowledge of the size in sequences and characters, and a sorting of the sequences before writing them back to file staring with the longest to shortest. The sorting was done to even out the workload distributed to each thread, preventing one thread from receiving all the long sequences and finishing much later than the others. The sorting only works if the workload is distributed in small chunks with the intention that all threads calculate a number of chunks and get a new chunk when they finish their task, thus the shorter, hence smaller tasks are given out at the end.

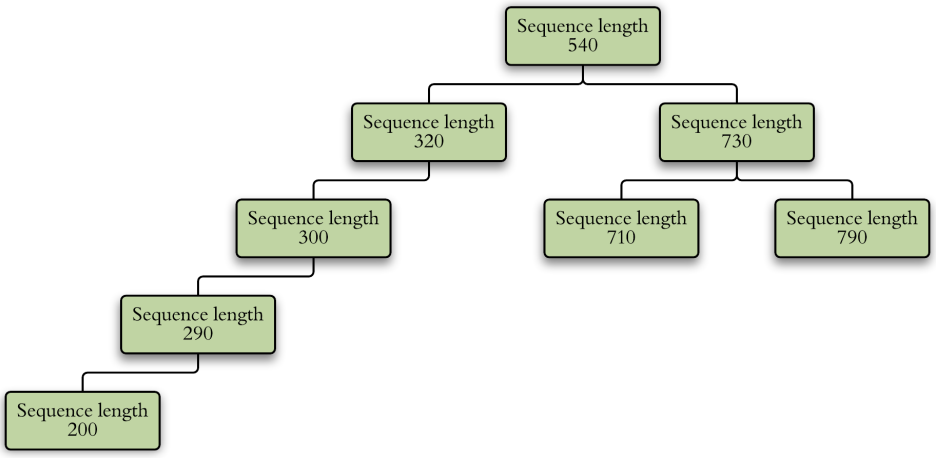

Figure 4.3: The binary tree structure used for sorting the database

For the sorting of the database a binary tree structure, shown in figure 4.3, was used. There was no need for a balanced tree since it is never used for searching, its only purpose was to keep the sequence order and write them

back to file in a recursive function starting form right to left, i.e. longest to shortest sequence.

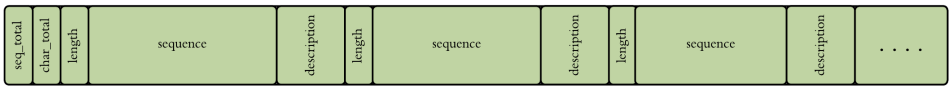

Figure 4.4: The processed database file

Figure 4.4 shows the processed database file with the total amount of sequences represented by a `long` as the first element, the total length of the database represented in `char`'s as the second element, and continuing through the entire database with a `long` that tells the length of the next sequence followed by the corresponding fixed length description.

The characters are mapped to a corresponding number between 0-27 before written to file. This to make the look ups in the substitution scoring matrix faster since there is no need to see what character it is, the score is now found by using the `int` value of the database entry for the `j` index and the `int` value of the query entry for the `i` index in the scoring matrix. The mapping is as Figure 4.5 shows.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
-	A	B	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	X	Y	Z	U	*	O	J

Figure 4.5: The mapping from characters to corresponding values

The preprocessed database is rapidly read into memory without reallocation and end of file (`EOF`) check since the required size is loaded first.

4.4.2 Distribution

To gain an optimal performance on the Xeon Phi, Section 2.2.3 presents some criteria that is useful to follow. Among them, the importance of conscious memory usage. To fulfill this requirement a distribution of the database was necessary to prevent each thread from extracting data form *16* different locations in memory when building a 512 bit vector of 16 32 bit `int` values. Without a change in the database this poor utilization of cache would have caused a huge performance bottleneck as a result of unnecessary memory reads.

The solution was to pair a given number N of sequences and write them to file with all the 16 first characters first, then the second character for all sequences and so on. When the end is reached for a sequence it is padded with '*' until the length of the longest of the N sequences is reached to ensure

that all N sequences is the same length and can be calculated in the same vector. Shown in Example 4.4.1

Example 4.4.1. An example with a simplified database with the following four same letter sequences.

F	F	F	F	F	F	F	F	F	F	F	F
M	M	M	M	M	M	M	M	M	M	M	
D	D	D	D	D	D	D	D	D	D		
Y	Y	Y	Y	Y	Y						

First they are padded to be the same length.

F	F	F	F	F	F	F	F	F	F	F	F
M	M	M	M	M	M	M	M	M	M	M	*
D	D	D	D	D	D	D	D	D	D	*	*
Y	Y	Y	Y	Y	Y	*	*	*	*	*	*

Then they are merged into one combined sequence with the first letter in all sequences, then the second letter and so on.

F	M	D	Y	F	M	D	Y	F	M	D	Y
F	M	D	Y	F	M	D	Y	F	M	D	Y
F	M	D	*	F	M	D	*	F	M	D	*
F	M	D	*	F	M	*	*	F	*	*	*

The padding is also included to make it possible to extract the initial sequences by assembling every fourth letter.

After N sequences and N descriptions are written, a new group starts. The length of the longest sequence in that group first, then the N next sequences padded and distributed and the corresponding descriptions. A drawing of the distributed database is shown in Figure 4.6.

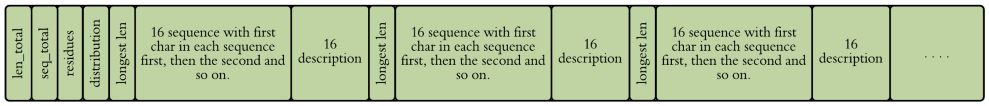

Figure 4.6: The distributed database file

The reason for two entries with the length of the database is because of the padding to make the N grouped sequences the same length, `residues` gives the correct number of residues in the database while the `len_total` includes the padded chars to give the right length for memory allocation. With this

modification the cached data is the next requested data by the application and the cache is utilized in an efficient and conscious matter.

4.5 Memory Management

To attain the conscious cache utilization prepared in the preprocessed database the loaded sequences are stored in a continuous `char` array like Figure 4.7 shows. The database array is allocated using the *memory aligned* `_mm_malloc` function to assure the required 64 byte memory alignment.

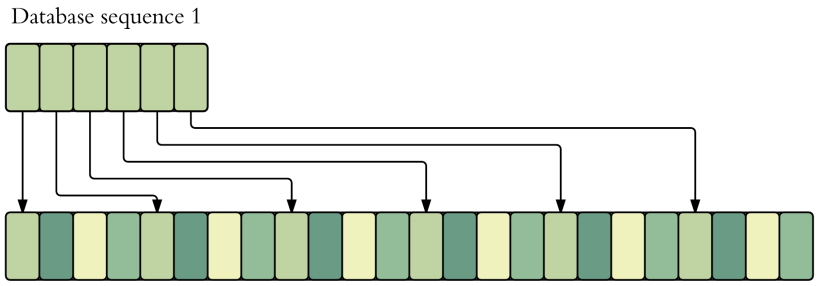

Figure 4.7: Database in memory

The remaining details about each sequence is stored in an array of the custom struct `idx_t`. Each sequence has its own entry with all required information such as the description and space to hold the later calculated score. The `idx_t` entry is also used to navigate the database array using the `index` variable stored in the entry. Figure 4.8 presents this structure with corresponding colors to Figure 4.7.

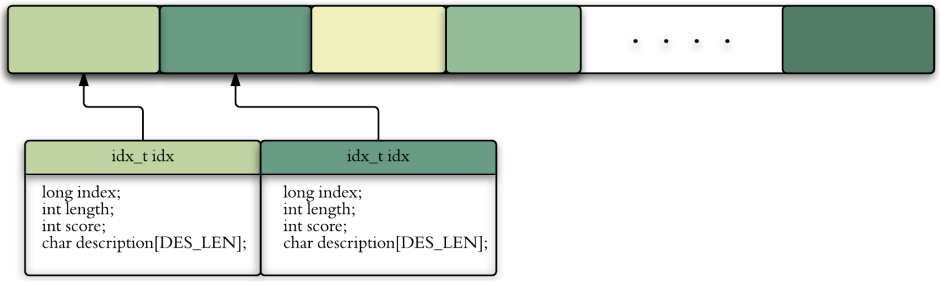

Figure 4.8: Table of contents for the database in memory

Since all database sequences has its own entry, there is no thread interfering while calculating the optimal alignment with the Smith-Waterman algorithm, hence no need for critical sections while executing and storing the alignment. This, in addition to memory aligned allocations, addresses well

with the memory criteria for gaining optimal performance in Section 2.2.3 on page 13.

4.5.1 Scoring Matrices

The scoring matrices are an important component in the alignment process. Every cell calculation in the H matrix requires a lookup in the scoring matrix, thus the importance of time and cache efficient retrieval of alignment scores. Due to the fact that the values in a scoring matrix never exceeds the highest or lowest value possible to represent with 8 bits, the scoring matrices are represented by a 64 byte aligned `int8_t` array.

Since the database and query sequences are mapped close to alphabetically to values as shown in Figure 4.5 on page 27, a alphabetically sorted scoring matrix makes for efficient lookups. The blosum62 matrix is sorted and illustrated in Figure 4.9. The included scoring matrices are `blosum45`, `blosum50`, `blosum62`, `blosum80` and `blosum90` in the blosum series. From the pam series `pam30`, `pam70` and `pam250` is included.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
	-	A	B	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	X	Y	Z	U	*	O	J
0 -	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1 A	-1	4	-2	0	-2	-1	-2	0	-2	-1	-1	-1	-1	-2	-1	-1	-1	1	0	0	-3	-1	-2	-1	-1	-4	-1	-1
2 B	-1	-2	4	-3	4	1	-3	-1	0	-3	0	-4	-3	4	-2	0	-1	0	-1	-3	-4	-1	-3	0	-1	-4	-1	-3
3 C	-1	0	-3	9	-3	-4	-2	-3	-3	-1	-3	-1	-1	-3	-3	-3	-3	-1	-1	-1	-2	-1	-2	-3	-1	-4	-1	-1
4 D	-1	-2	4	-3	6	2	-3	-1	-1	-3	-1	-4	-3	1	-1	0	-2	0	-1	-3	-4	-1	-3	1	-1	-4	-1	-3
5 E	-1	-1	1	-4	2	5	-3	-2	0	-3	1	-3	-2	0	-1	2	0	0	-1	-2	-3	-1	-2	4	-1	-4	-1	-3
6 F	-1	-2	-3	-2	-3	-3	6	-3	-1	0	-3	0	0	-3	-4	-3	-3	-2	-2	-1	1	-1	3	-3	-1	-4	-1	0
7 G	-1	0	-1	-3	-1	-2	-3	6	-2	-4	-2	-4	-3	0	-2	-2	-2	0	-2	-3	-2	-1	-3	-2	-1	-4	-1	-4
8 H	-1	-2	0	-3	-1	0	-1	-2	8	-3	-1	-3	-2	1	-2	0	0	-1	-2	-3	-2	-1	2	0	-1	-4	-1	-3
9 I	-1	-1	-3	-1	-3	-3	0	-4	-3	4	-3	2	1	-3	-3	-3	-3	-2	-1	3	-3	-1	-1	-3	-1	-4	-1	3
10 K	-1	-1	0	-3	-1	1	-3	-2	-1	-3	5	-2	-1	0	-1	1	2	0	-1	-2	-3	-1	-2	1	-1	-4	-1	-3
11 L	-1	-1	-4	-1	-4	-3	0	-4	-3	2	-2	4	2	-3	-3	-2	-2	-2	-1	1	-2	-1	-1	-3	-1	-4	-1	3
12 M	-1	-1	-3	-1	-3	-2	0	-3	-2	1	-1	2	5	-2	-2	0	-1	-1	-1	1	-1	-1	-1	-1	-1	-4	-1	2
13 N	-1	-2	4	-3	1	0	-3	0	1	-3	0	-3	-2	6	-2	0	0	1	0	-3	-4	-1	-2	0	-1	-4	-1	-3
14 P	-1	-1	-2	-3	-1	-1	-4	-2	-2	-3	-1	-3	-2	-2	7	-1	-2	-1	-1	-2	-4	-1	-3	-1	-1	-4	-1	-3
15 Q	-1	-1	0	-3	0	2	-3	-2	0	-3	1	-2	0	0	-1	5	1	0	-1	-2	-2	-1	-1	4	-1	-4	-1	-2
16 R	-1	-1	-1	-3	-2	0	-3	-2	0	-3	2	-2	-1	0	-2	1	5	-1	-1	-3	-3	-1	-2	0	-1	-4	-1	-2
17 S	-1	1	0	-1	0	0	-2	0	-1	-2	0	-2	-1	1	-1	0	-1	4	1	-2	-3	-1	-2	0	-1	-4	-1	-2
18 T	-1	0	-1	-1	-1	-1	-2	-2	-2	-1	-1	-1	-1	0	-1	-1	-1	1	5	0	-2	-1	-2	-1	-1	-4	-1	-1
19 V	-1	0	-3	-1	-3	-2	-1	-3	-3	3	-2	1	1	-3	-2	-2	-3	-2	0	4	-3	-1	-1	-2	-1	-4	-1	2
20 W	-1	-3	-4	-2	-4	-3	1	-2	-2	-3	-3	-2	-1	-4	-4	-2	-3	-3	-2	-3	11	-1	2	-2	-1	-4	-1	-2
21 X	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
22 Y	-1	-2	-3	-2	-3	-2	3	-3	2	-1	-2	-1	-1	-2	-3	-1	-2	-2	-2	-1	2	-1	7	-2	-1	-4	-1	-1
23 Z	-1	-1	0	-3	1	4	-3	-2	0	-3	1	-3	-1	0	-1	4	0	0	-1	-2	-2	-1	-2	4	-1	-4	-1	-3
24 U	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-4	-1	-1
25 *	-1	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	1	-1	-4
26 O	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
27 J	-1	-1	-3	-1	-3	-3	0	-4	-3	3	-3	3	2	-3	-3	-2	-2	-2	-1	2	-2	-1	-1	-3	-1	-4	-1	-3

Figure 4.9: The scoring matrix blosum62 sorted alphabetically and mapped to corresponding values

4.6 Threading

The most efficient way to utilize OpenMP is to set up the work to be done in a for loop and let the OpenMP API distribute the workload, either statically by giving a fixed chunk to each thread or dynamically where the distribution is done at runtime. For SWIMIC the most promising way to set up the workload was to use a for loop iterating through the sequences and let OpenMP distribute the inner calculations dynamically, hence the sorting of the database sequences from longest to shortest to avoid a large workload being distributed to one thread at the end of the calculation and when all other threads are finish they have to wait for the last one. With this workload distribution the scalability to more than a hundred threads is good and the scalability criteria from Section 2.2.3 is fulfilled.

4.6.1 Affinity

OpenMP has some built-in features on how to distribute the workload between cores; *compact*, *scatter*, *balanced* and *none*. Since there is 4 hardware threads per core it may make a difference in what order the tasks are divided among the cores. As Figure 4.10 shows, *scatter*, *compact* and *balanced* have a predefined pattern on how to distribute, while *none* just randomly distributes the tasks. In this simple example only 8 of 16 threads where used, but the same principles applies when all threads are contributing. If the tasks distributed are independent on each other, a *scatter* or *balanced* affinity is more likely to be efficient. On the other hand, if the tasks use a lot of the same data, a *compact* affinity may yield a better result, since closely cached data is more quickly retrievable.

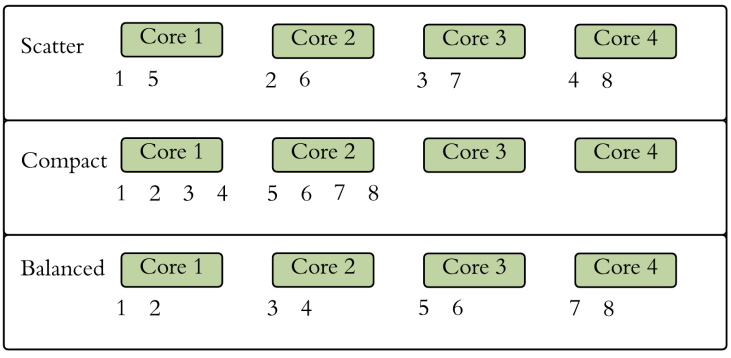

Figure 4.10: OpenMP affinity distribution

The thread affinity interface of the Intel runtime library can be controlled by using the `KMP_AFFINITY` environment variable.

4.7 Vectorization

To fully utilize the Xeon Phi architecture, an effective and well thought out implementation is needed. The most unique feature is the 512 bit vector unit which provides an exceptional opportunity to perform the same action to a large multiple of sequences at once. The desirable outcome of vectorization on the Xeon Phi has to be able to triple the number of actions performed simultaneously compared to a regular CPU with a 128 bit vector unit, hence a close to triple speed gain. Swipe [4] uses SIMD intrinsics to perform 16 smith-waterman search simultaneously by using only 8 bit of the 128 bit vector unit available in a regular CPU per database sequence. Because of the small space given to each sequence, overflow/underflow is unfortunately a problem and needs to be accounted for.

When looking through the list of the Xeon Phi's intrinsics it was like a bucket of cold water was thrown in this thesis direction. The Xeon Phi did not support operations on smaller data types than `int`, e.i. 32 bit! That meant the end for the anticipated threefold increase in actions and speed, since dividing the 512 bit vector unit into 32 bit only adds up to 16, which is the same multiple of sequences as Swipe.

On a slight positive, the need for overflow/underflow check when utilizing calculations on 8 bit disappeared, since the score for an alignment would never exceed the `INT_MAX` value of 32767. However, this is nothing compared to the performance gain lost by the unsupported smaller data types.

The algorithm to vectorize is the Smith-Waterman algorithm [2] with Gotoh's [15] modification discussed in section 3.1.2. The algorithm is included below to refresh the memory and to look at when the steps in the algorithm is vectorized.

There are four main steps in the algorithm:

- Find the score from the scoring matrix corresponding to the query and the database sequence being aligned, and add the score to H.
- Find max of H, E and F, and make sure it is not a negative number.
- Save score in S if the score is higher than previously calculated max
- Update H, E, and F to reflect potential gap openings and/or extensions.

$$H_{i,j} = \begin{cases} \max \begin{cases} H_{i-1,j-1} + SM[q_i, d_j] \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} & \begin{matrix} i > 0 \\ \cap \\ j > 0 \end{matrix} \\ 0 & \begin{matrix} i = 0 \\ \cup \\ j = 0 \end{matrix} \end{cases}$$

$$E_{i,j} = \begin{cases} \max \begin{cases} H_{i,j-1} - Q \\ E_{i,j-1} - R \end{cases} & j > 0 \\ 0 & j = 0 \end{cases}$$

$$F_{i,j} = \begin{cases} \max \begin{cases} H_{i-1,j} - Q \\ F_{i-1,j} - R \end{cases} & i > 0 \\ 0 & i = 0 \end{cases}$$

$$S = \max_{1 \leq i \leq m \cap 1 \leq j \leq n} H_{i,j}$$

Since E and F holds the value for the previous step, the calculation of gaps can be performed as the last step for the current entry to exploit the cache. A simplified SIMD vectorization is shown in Figure 4.11 using vectorized add and subtraction along with the ability to find the maximum value of each element in two vectors, all simple instructions the coprocessor possesses.

The reason why a for loop is used for setting up the score matrix vector (SM) instead of the `_mm512_set_epi32` intrinsic is due the fact that auto vectorization of loops on the Xeon Phi proves to be faster then the `set` function.

For an easier and more flexible way of doing the alignment a define was created, shown in Figure 4.12

4.7.1 Additional Alignments

By increasing the number of database sequences aligned per character in the query, the utilization of local cache got expanded. Figure 4.13 illustrates the alignment with additional database sequences being aligned against the same query sequence simultaneously.

To be able to utilize the same function as earlier, and having a conscious cache utilization, an additional padding and distribution of 32 and 64 sequences was necessary. This was a simple task, the problem was how

```
// -------- step 1 ------------

for (t = 0; t < SW_MULTIPLE; t++) {
    tmp[t] = score_matrix(db, query);
}
SM = *(__m512i*)tmp;
H = _mm512_add_epi32(H,SM);

// -------- step 2 ------------

H = _mm512_max_epi32(H,E);
H = _mm512_max_epi32(H,F);
H = _mm512_max_epi32(H,zero);

// -------- step 3 ------------

S = _mm512_max_epi32(H,S);

// -------- step 4 ------------

E = _mm512_sub_epi32(E,GAPEXTEND);
F = _mm512_sub_epi32(F,GAPEXTEND);
H = _mm512_sub_epi32(H,GAPOPENEXTEND);
E = _mm512_max_epi32(H,E);
F = _mm512_max_epi32(H,F);
```

Figure 4.11: SIMD code

```
#define ALIGN(H, N, E, F, SM, S)                              \
H = _mmx_add_epi32(H,SM);    /* add comparability score to H */\
H = _mmx_max_epi32(H,F);     /* MAX (H, F)                   */\
H = _mmx_max_epi32(H,zero);  /* Make sure H > 0              */\
H = _mmx_max_epi32(H,E);     /* MAX (H, E)                   */\
S = _mmx_max_epi32(H,S);     /* Save max score               */\
N = H;                       /* Save H for next step         */\
H = _mmx_sub_epi32(H,GOE);   /* SUB gap open-extend          */\
F = _mmx_sub_epi32(F,GE);    /* SUB gap extend               */\
F = _mmx_max_epi32(H,F);     /* Test for opening or extend   */\
E = _mmx_sub_epi32(E,GE);    /* SUB gap extend               */\
E = _mmx_max_epi32(H,E);     /* Teat for opening or extend   */
```

Figure 4.12: SIMD ALIGN code

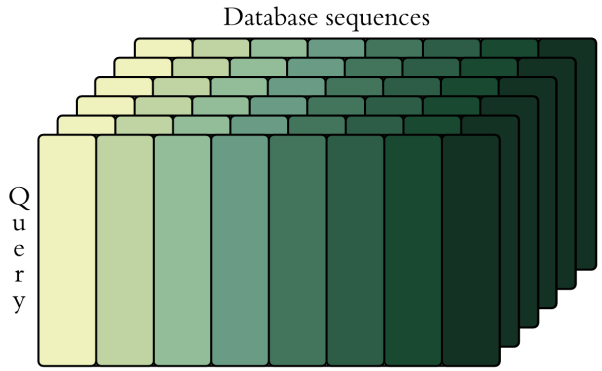

Figure 4.13: Additional alignments of database sequences per iteration

```
for (t = 0; t < SW_MULTIPLE; t++) {
    tmp[t] = score_matrix(db, query);
}
SM = *(__m512i*)tmp;
ALIGN(H0, N2, E, F0, SM, S);

for (t = 0; t < SW_MULTIPLE; t++) {
    tmp[t] = score_matrix(db+x , query);
}
SM = *(__m512i*)tmp;
ALIGN(H1, N1, E, F1, SM, S);
```

Figure 4.14: SIMD code for multiple alignments per iteration.

much padding was added and did the extra calculations on the padded areas increased the runtime more than the cache utilization gained? Both a solution for two and four times more database sequences were implemented. A code example with two times more sequences are shown in Figure 4.14. The `ALIGN` function is the one shown in Figure 4.12, and the x added to the database in the second for loop represents getting the next 16 database sequences, thus giving `SW_MULTIPLE` more sequences to align in this iteration.

4.7.2 Multiple Columns

An optimization used in Swipe [4] is to calculate four columns per iteration, so a natural experiment was to try this out on the Xeon Phi to see if this would give a speed gain on the coprocessor as well.

Starting from the first row, a range of x columns, i.e. x number of characters in the database sequences, are calculated for the entire query

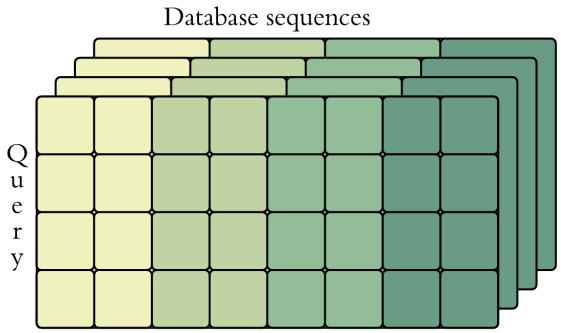

Figure 4.15: Illustration of calculating a range of columns

sequence. In Figure 4.15 the range is 2 and the coloring shows the order of calculations. All matrix element of the same color is calculated before the next color is carried out.

The code for doing this is the same as the one shown in Figure 4.14, with the exception of what the x represents. Now the x represent getting the next character in the database sequences already carried out this iteration.

With this approach there is less need to cache in between values, which gives more room in the local cache for additional calculations.

4.8 Pragma Directives

When programming one can use a pragma to specifie how a compiler should process its input. In some cases pragmas specify global behavior, while in other cases they only affect a local section, such as a block of programming code. Relevant in this thesis are the pragmas concerning parallelization, vectorization and memory management.

For distribution of the workload the OpenMP directive is used. As mentioned in Section 4.3, `#pragma omp for schedule(dynamic)` will distribute the workload in the following loop dynamically at runtime.

To make the loop run in parallel one may add `parallel` before `for` in the already existing pragma. However this will only work if all work supposed to run in parallel is inside the loop. If for instance a memory allocation is required for each thread running in parallel it is not optimal to do this inside the loop, making redundant allocations. A better approaches is to use `#pragma parallel` that will make a parallel section where the for loop may reside inside. This is exactly how SWIMIC is implemented.

When it comes to vectorization the compiler auto vectorize wherever it can, however this may not apply to all suitable sections if the compiler is unsure of vector alignment. The `#pragma vector` indicates that the loop should be vectorized, if it is legal to do so, ignoring normal heuristic decisions

about profitability. In terms of alignment of data adding `#pragma vector aligned` informs the compiler that all data is aligned to the required 64 byte, thus no checks for alignment is done. This may cause incorrect data if the memory is not alignment, but since all memory allocations in SWIMIC are using `_mm_malloc` this is should never be a problem.

4.9 Match Handling

To hold on to the N sequences that has the best match a custom array list was used. Each array element is as shown in Figure 4.16, and contain the score of the matching sequence and its database index, in addition to the pointers to the next and previous elements according to the sorting from highest to lowest score.

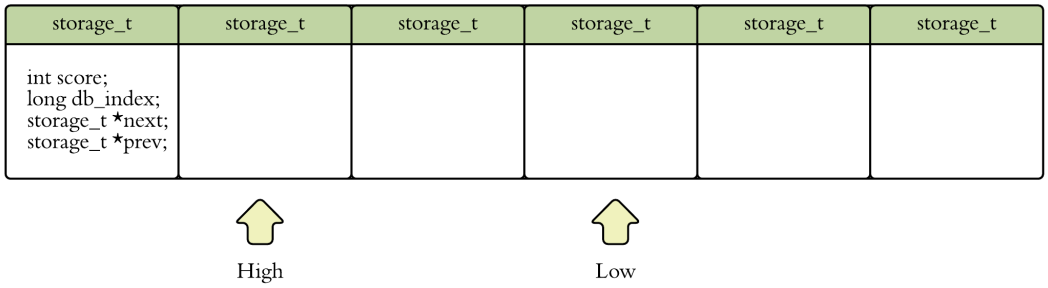

Figure 4.16: Storage structure for the best matching sequences

To have the memory allocated as an array makes it easy to access each element and then to combine it with pointers from highest to lowest element to have it as a sorted list as well provides a convenient advantage.

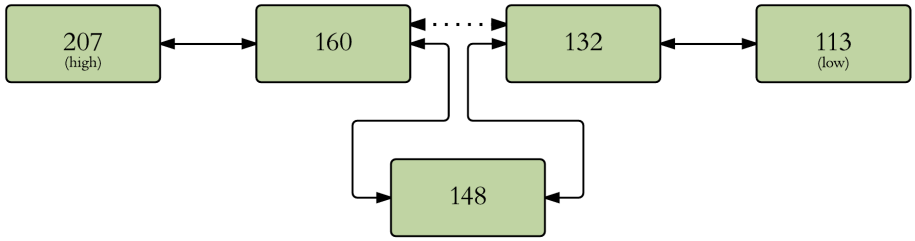

Figure 4.17: Insertion in a sorted doubly linked list

The N array elements only change content when a new match that is higher than the lowest saved score is found. First the score is compared to the score pointed to by low, if it is higher, the content in the array element pointed to by low is updated to reflect the new sequence. To attain the sorting

the "new" element is compared to each element from the highest and added as shown in Figure 4.17. The pointers are updated as a common doubly linked list, and the dotted line in the figure is before the new element is added while the solid lines represents the pointers after the insertion.

4.10 Parameters

To run SWIMIC with default settings the only parameters required is `-d`, a preprocessed database file in FASTA format and, `-q`, a query also in FASTA. Other options are shown in Figure 4.18. The values in parentheses are the default option.

```
Usage: ./sw_s [OPTIONS]
  -h, --help               show help
  -d, --db=FILE            sequence database filename (required)
  -q, --query=FILE         query sequence filename (required)
  -s, --matrix=NAME/FILE   score matrix name or filename (BLOSUM62)
  -t, --threads            number of threads (120)
  -r, --range              number of columns calculated per iteration (10)
  -a, --affinity           the affinity used by OpenMP (balanced)
  -m, --matches            number stored matches (20)
```

Figure 4.18: SWIMIC's parameters.

4.11 Output

The output from SWIMIC is divided into two parts. The first is a header with information about the database, query and alignment specifications.

The second part contains the alignment in a human readable format. Each match starts with the calculated alignment score and the description. Due to limited memory on the Xeon Phi, only 30 characters are saved, however the unique sequence specifier is included in the 30 characters. Following is a representation of the alignment in three lines. The first is the database sequence, the second showcases a symboled mapping of the alignment, and the third line is the query. A "-" in the database sequence or query indicates an insertions or deletion. From the symbol line insertions are indicated by a "+" and deletions are represented by "-". If it is a pairwise perfect match between the database sequence and the query, the matching character is written in the symboled line as well.

An example output is shown in Figure 4.19. The query sequence and the alignment representation is shortened down to to a minimum to showcase

a representative preview. Match 2 - 8 is also discarded in the example for a simpler, yet cleaner overview.

```
Database size    : 171625029 residues in 459313 sequences
Query file name  : P20930
Query lengt      : 4061
Score matrix     : blosum62
Gap penalty      : 11 + 1
Max matches      : 10
Threads          : 240
Elapsed          : 16.207916 s
GCUPS            : 43.001780


Query sequence   :
MSTLLENIFAIINLFKQYSKKDKNTDTLSKK ...

1: Sequence aligned with score 21359
Description: sp|P20930.3|FILA_HUMAN RecName ...

DB:  MSTLLENIFAIINLFKQYSKKDKNTDTLSKKELKELLEKEFRQILKNPDDPDMVDVFMDHLDIDH
SYM: MSTLLENIFAIINLFKQYSKKDKNTDTLSKKELKELLEKEFRQILKNPDDPDMVDVFMDHLDIDH
Q:   MSTLLENIFAIINLFKQYSKKDKNTDTLSKKELKELLEKEFRQILKNPDDPDMVDVFMDHLDIDH

...

9: Sequence aligned with score 685
Description: sp|Q9NZW4.2|DSPP_HUMAN RecName ...

DB:  KES-GVLVHEGDR-GRQENTQDG-HKGEGNGSKWAEV--GG-KSFSTYSTLANEEGNIEGWNGDT
SYM:   S+G  -H-G  +- Q -    +H G G G     V++ G+   S      NE-G  E--  DT
Q:   RSSAGER-H-GSHH-QQS-ADSSRHSGIGHGQASSAVRDSGHRGYSGSQASDNE-GHSE--DSDT
```

Figure 4.19: A simplified overview of SWIMIC's output.

Chapter 5

Result and Discussion

This chapter presents and discusses the results developed and the experiences learned during the implementation process. Some key aspects are implementation testing, comparison to other tools and exploitation of the coprocessor.

5.1 Implementation Testing

To examine the performance of the implemented tool on the Xeon Phi coprocessor a number of tests where performed with different parameters. Some to exploit how well the features on the coprocessor is utilized while others examined different parallelization approaches.

The common unit of measure in bioinformatics GCUPS (billion cell updates per second) is used to determine the performance of the alignment tool in this thesis. CUPS represents the time for a complete computation of one cell in the H matrix, including all memory operations and the corresponding computation of the values in the E and F matrices.

Some tests only showed a slight differences in performance, while others made a huge impact. Some of the results dose not showcase the finished results, but presents the result from the same code at a given stage with only one difference that make a huge leap in performance. This to avoid unnecessary work on code that are discarded at an early stage in the optimizing process.

To make the testing process as simple and efficient as possible, a handful of scripts where made to easily redo the tests if a change was made in the source code. In the early stages of the implementation process the interesting aspects to look into was the number of threads required for the best performance. Later on the necessity to test for a variety of query lengths became more important. To acquire reliable test results, an average of 10 executions with the same parameter are calculated for all queries.

5.1.1 Test Conditions

The Xeon Phis used for testing in this thesis are 4 nodes with two 60 core mic processors with a 8 GiB on card memory each. The exact specifications from `/proc/cpuinfo` is shown below for one of the cores. The Xeon Phis are provided by the Abel Cluster owned by the University of Oslo. Additionally a Xeon Phi card provided by SINTEF was to be used. The setting up process is described in Appendix A.

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 11
model           : 1
model name      : 0b/01
stepping        : 3
cpu MHz         : 1052.630
cache size      : 512 KB
physical id     : 0
siblings        : 240
core id         : 59
cpu cores       : 60
apicid          : 236
initial apicid  : 236
fpu             : yes
fpu_exception   : yes
cpuid level     : 4
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8
                  apic mtrr mca pat fxsr ht syscall nx
                  lm nopl lahf_lm
bogomips        : 2094.71
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
```

For time measure the `gettimeofday()` function from the `time.h` library is used. The timing starts at the first call to the Smith-Waterman algorithm and ends when the desired number of matches is sorted and saved.

The database used for testing is the UniProtKB/Swiss-Prot [7] database. The version of the database used contained 171625029 residues in 459313 sequences.

To examine how SWIMIC's performance depends on different query lengths a handful of sequences with various length from the UniProtKB/Swiss-Prot database where chosen. The sequences chosen are the same sequences

Query name	length	Query name	length	Query name	length
P03630	127	P58229	511	P04775	2005
P02232	144	P25705	553	P19096	2504
P01111	189	P03435	567	P28167	3005
P14942	222	P42357	657	P0C6B8	3564
P00762	246	P21177	729	P20930	4061
P53765	255	O60341	852	P08519	4548
P03989	362	P0A621	934	Q7TMA5	4743
P07327	375	P27895	1000	P33450	5147
P01008	464	P9WPK2	1115	Q9UKN1	5478
P10635	497	P07756	1500		

Table 5.1: The queries used to test the implementation

used to test SWIPE, with a few exceptions. Table 5.1 presents the sequences with unique names and corresponding lengths. Figure 5.1 presents the results from an execution with the default parameters. These are the `blosum62` scoring matrix, a range of 10, 240 threads and a `balanced` affinity. By default SWIMIC executes four threads per core, 240 in total. This is being substantiated by the plot over different number of threads presented in Figure 5.2.

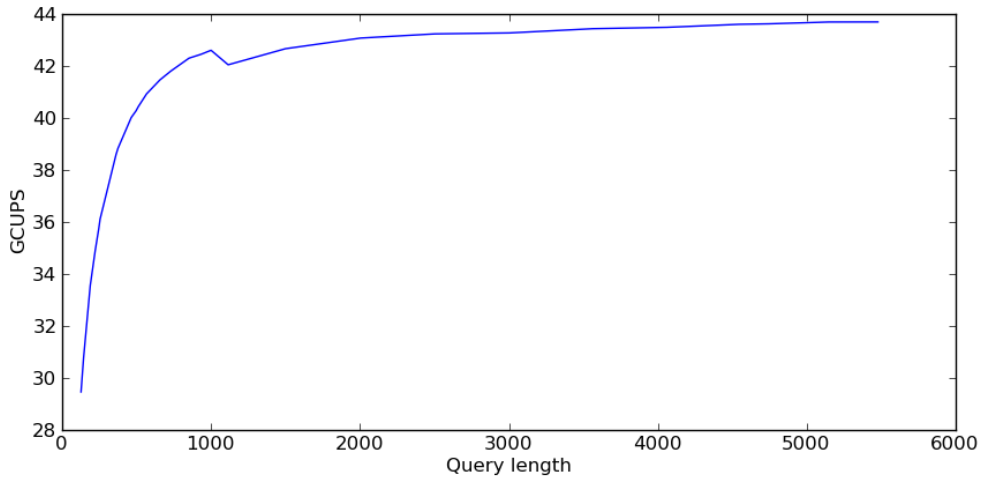

Figure 5.1: GCUPS achieved with different query lengths with 240 threads

5.1.2 Memory Management

From Section 2.2.3's list of criteria to consider when optimizing for the Xeon Phi coprocessor cache usage was listed. A couple of test to illustrate this effect

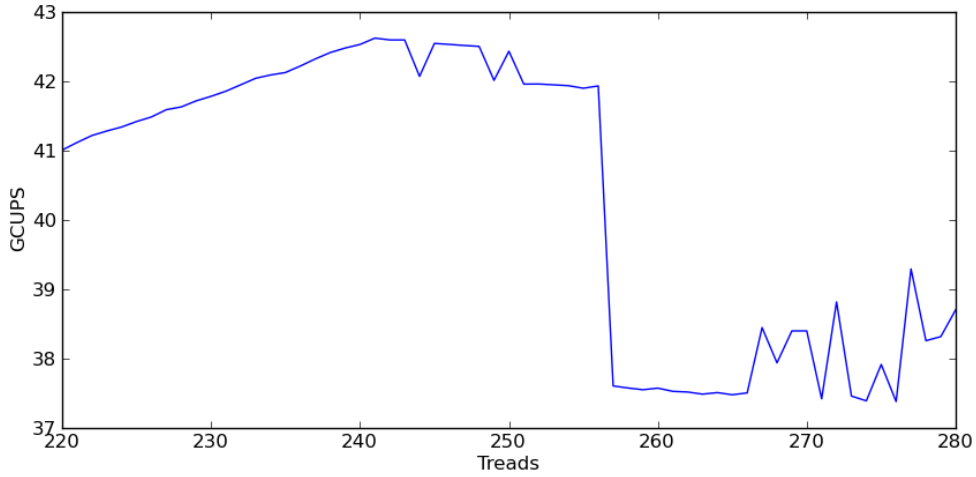

Figure 5.2: GCUPS achieved with different number of threads with a 1000 amino acids long sequence

is shown below.

Memory Location

A slight code change from bad to better cache utilization that made a huge difference in performance is how to do lookups in the scoring matrix. Since the scoring matrix is squared, and the values are the same in both direction, it does not matter if it is the query or the database sequence that are represented horizontally or vertically. However performance wise, this differences matters a lot. Consider the two lines of code:

```
tmp = score_matrix[db[i]*32 + query[j]];

tmp = score_matrix[query[j]*32 + db[i]];
```

Both lines gives the same result, the only difference is where data is read from memory. Since the same query are compared to a multiple of sequences the data required in each iteration is the data on the same column/row. With the first line the data is read from memory as shown in the first figure in Figure 5.3, where all data needed is stored down a column. This causes redundant caching since the scoring matrix is stored row wise in memory, thus the next data needed is not the next data in memory. With the second line the data is read as a row and the next data needed is saved in cache.

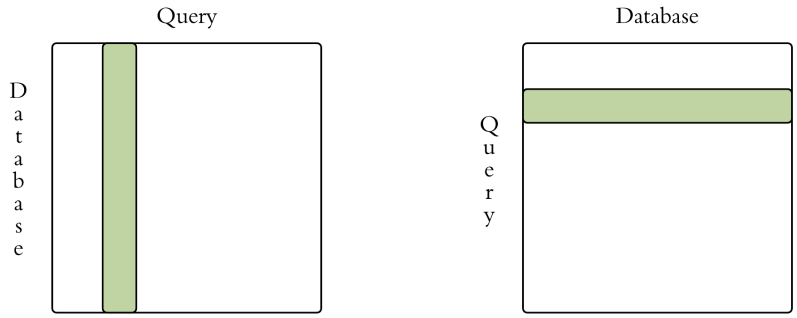

Figure 5.3: Memory reads

To illustrate the huge difference in performance Figure 5.4 gives a good view combined with Table 5.2.

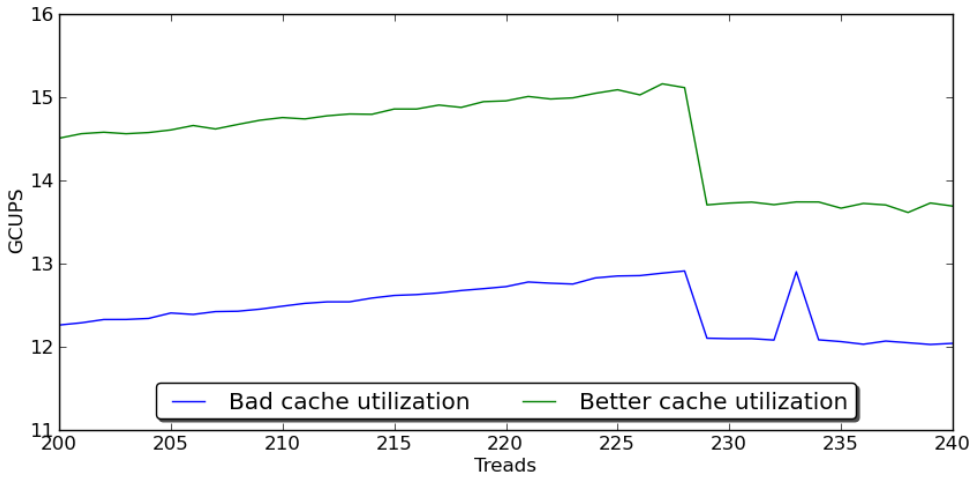

Figure 5.4: Comparison of lookups in scoring matrix

Threads used	200	220	228	240
Bad cache utilization	12.26	12.72	**12.91**	12.04
Better cache utilization	14.50	14.95	**15.11**	13.69

Table 5.2: Comparison of cache utilization

Distributed Database

As described in Section 4.4.2 the database was distributed to achieve a superior cache utilization. A plot to accentuate the performance impact is

presented in Figure 5.5 with the corresponding values in Table 5.3.

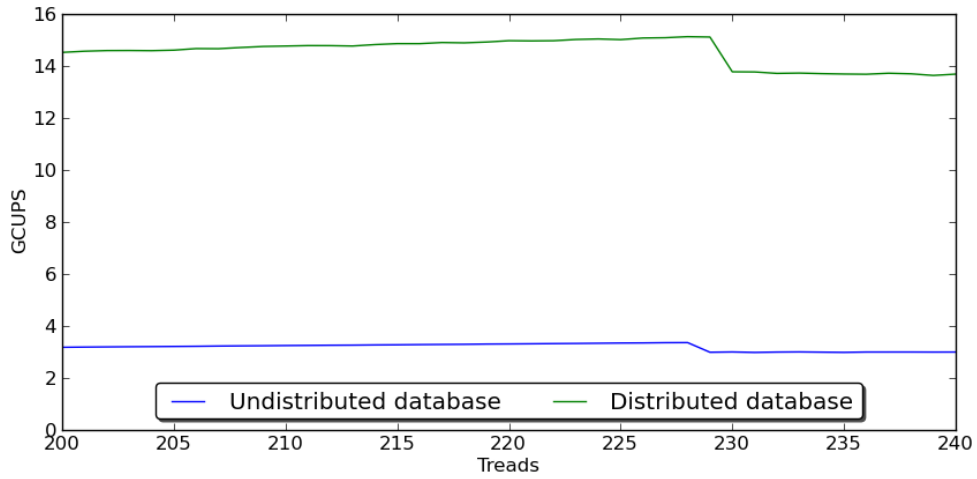

Figure 5.5: Comparison between a distributed and an undistributed database

Threads used	200	220	228	240
Undistributed database	3.180	3.309	**3.361**	2.997
Distributed database	14.51	14.95	**15.13**	13.70

Table 5.3: Comparison between a distributed and an undistributed database

As expected the undistributed database performs far below what the distributed database achieves. Without the distribution the implementation is most likely doing a plethora of unnecessary memory reads for each lookup in the scoring matrix. The cached data are useless for the rest of the iteration step and is almost immediately swapped out to make room for more prominent data. This poor performance is due to a worst case scenario of 15 additional memory reads per character in each database sequence, all with an average sequence length of 300-400 characters.

The comparison was performed at an early stage in the optimization process and do not showcase an optimized implementation. The only difference between the to executions in the plot in Figure 5.5 is how the database is distributed.

5.1.3 Additional Alignments

In section 4.7.1 an alternative implementation of vectorization with multiple vectors to align additional database sequences per iteration was presented. Figure 5.6 and Table 5.4 shows the results of a comparison test between one,

two and four vectors. Interesting to notice is that with more vectors, the use of fewer threads yields the highest result, due to higher memory usage per thread. What is surprising is that the use of two vectors never comes close to the peak point for either one or four vectors.

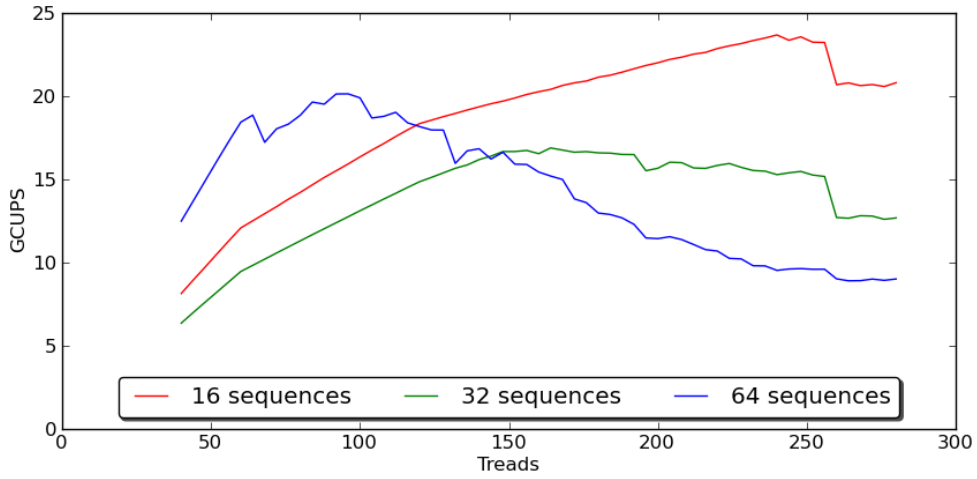

Figure 5.6: Comparison of one, two and four additional vectors

Threads used	96	120	164	200	240
16 sequences	15.92	18.35	20.40	22.00	**23.68**
32 sequences	12.75	14.85	**16.89**	15.67	15.28
64 sequences	**20.14**	18.17	15.20	11.44	9.531

Table 5.4: Performance with additional vector calculations for a variety of number of threads

This approach did not exceed the performance of one vector and was thus abandoned. A combination with the approaches in the next section may have resulted in a better optimization, but due to time constrains this task have been left for future work.

5.1.4 Multiple Columns

Since aligning multiple columns simultaneously is a rewarding optimization for SWIPE [4] this was one of the most exciting optimizations to examine. SWIPE uses a range of 4 columns to align simultaneously, while for SWIMIC a variety of ranges where tested. Figure 5.7 with the corresponding values in Table 5.5 shows that the performance gain from only four columns is not

sufficient on the Xeon Phi coprocessor. A range as wide as 10 columns is what yields the best performance for all query lengths.

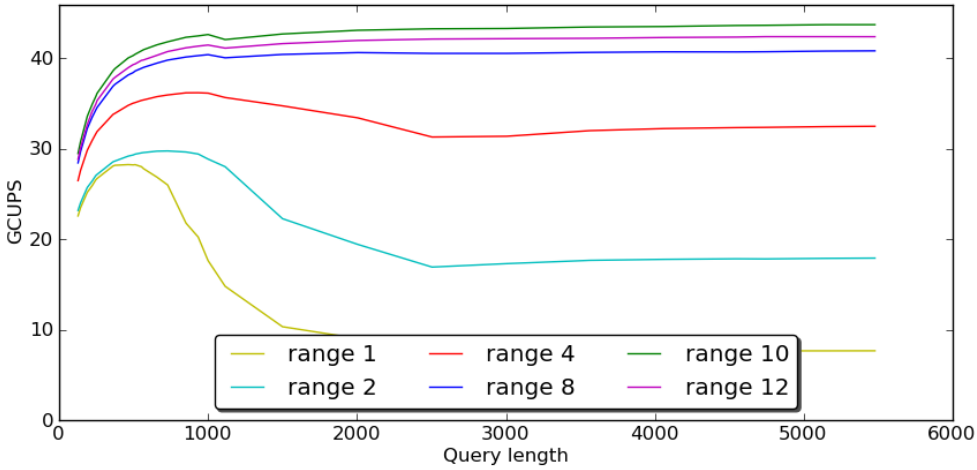

Figure 5.7: Comparison of different column ranges

Query length	189	567	1000	2005	5147
Range 1	25.167	27.783	17.627	9.015	7.654
Range 4	29.812	35.376	36.128	33.398	32.427
Range 8	32.221	38.975	40.381	40.609	40.773
Range 10	33.561	40.927	42.604	43.074	43.691
Range 12	32.667	39.795	41.442	41.946	42.369

Table 5.5: GCUPS achieved for a variety of query lenghts with different column range

5.1.5 Scoring Matrices

To exclude deviation between different scoring matrices a comparison between all SWIMIC's included scoring matrices is shown in Figure 5.8. From Table 5.6 one can see that the values are almost identical for all matrices and the possible variation in performance is excluded.

5.1.6 Match Handling

The process of saving and sorting sequences that matches the query may slow down the performance. Especially since the implemented structure

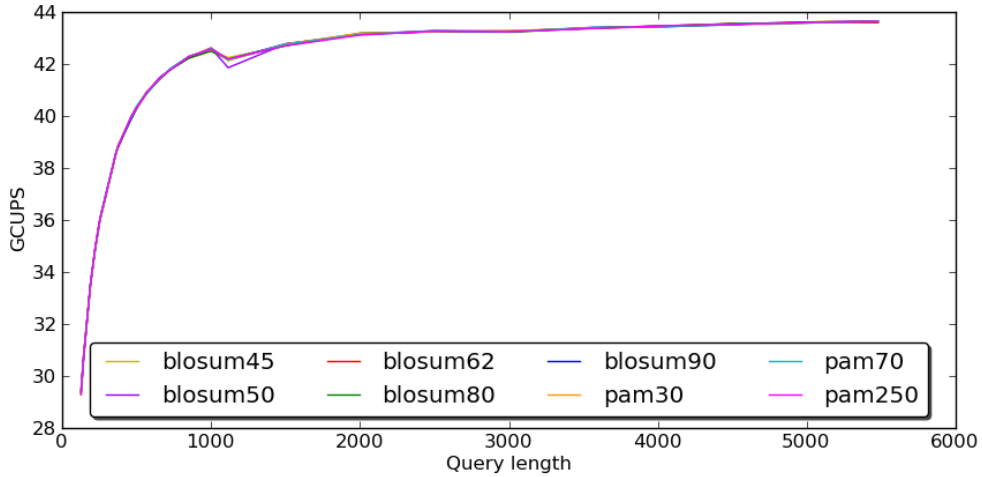

Figure 5.8: Scoring matrices comparison

Query length	189	567	1000	2005	5147
Blosum45	33.438	40.937	42.514	43.169	43.643
Blosum50	33.447	40.898	42.612	43.142	43.622
Blosum62	33.520	40.859	42.617	43.156	43.606
Blosum80	33.477	40.899	42.489	43.192	43.605
Blosum90	33.481	40.884	42.587	43.136	43.612
Pam30	33.437	40.878	42.600	43.185	43.603
Pam70	33.476	40.843	42.594	43.138	43.593
Pam250	33.418	40.895	42.579	43.115	43.609

Table 5.6: The GCUPS score for different query lenght

is straightforward and not resting on predefined libraries. To validate the structure and remove all doubt of the performance Figure 5.9 shows that the time used in the sorting process is a drop in the ocean when it comes to performance compared to the alignment process.

5.1.7 Affinity

Table 5.7 with the corresponding plot, Figure 5.10, shows that for SWIMIC the affinity, described in Section 4.6.1, does not make a difference. As long as one does not exceed the maximum number of threads per core, in this case 240, all affinities gives the same result. With a larger number of threads there is a considerable variation in performance until the cache usage exceeds its maximum and the cost of reading more data from memory downgrades the performance.

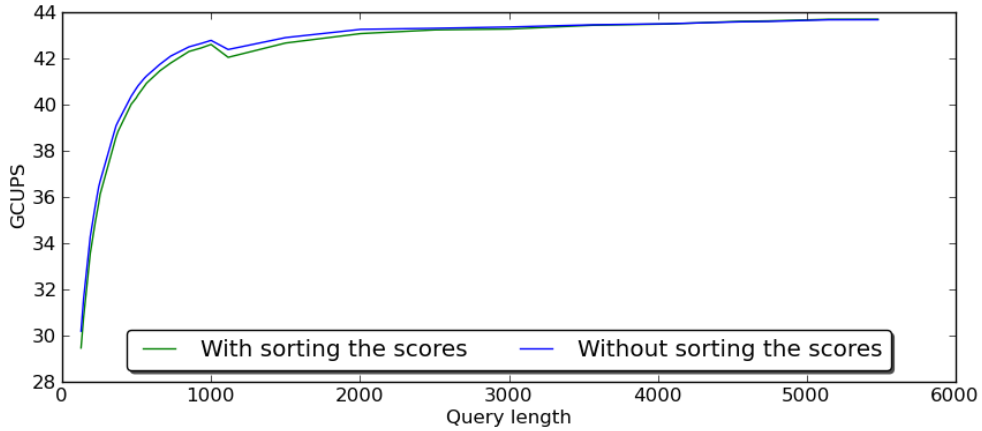

Figure 5.9: Performance without sorting the calculated score

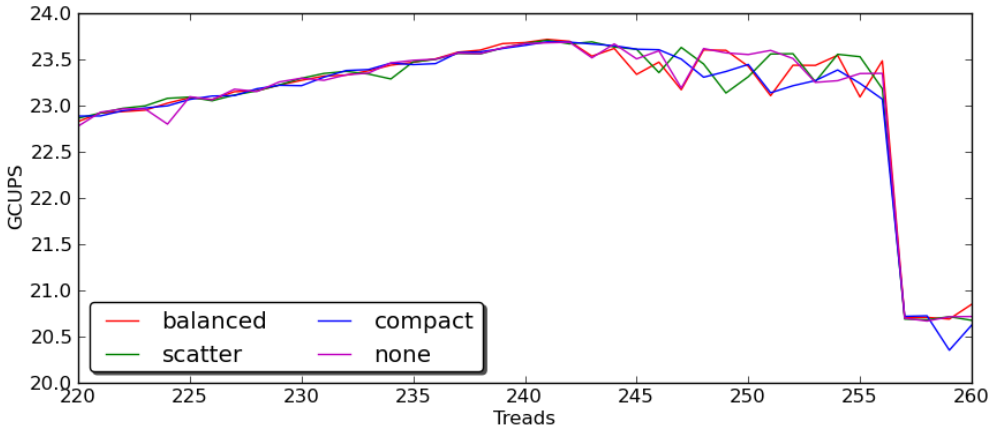

Figure 5.10: Comparison of the different built-in OpenMP thread affinities

Threads used	220	240	241	260
balanced	22.82	23.68	**23.71**	20.84
scatter	22.88	23.65	**23.69**	20.62
compact	22.86	23.67	**23.70**	20.67
none	22.77	23.67	**23.67**	20.71

Table 5.7: Comparison of the different built-in OpenMP thread affinities

Due to this a `balanced` affinity is set to default for SWIMIC.

5.2 Performance Analysis

Unfortunately VTune, the profiler from Intel that was going to help the analysis process, failed to install correctly. After numerous attempts and troubleshooting the profiler had to be abandoned. This meant that much of the information automatically given from the profiler now had to be examined manually. This included finding bottlenecks in both memory usage and CPU performance. In addition appropriate solution to these bottlenecks had to be manually examined.

vec-report from the Compiler

The Intel compiler includes an option to give a vectorization report using the flag `-vec-report[1-6]` With this option a file with *.optrpt* extension is created for each .c file. In this report each loop in the code is evaluated and remarks are written. A code snippet from the report is presented in Figure 5.11. The loop evaluated in this example is one of the for loops in the code form Figure 4.14 on page 35.

```
LOOP BEGIN at SIMD.c(604,13)
   remark #15388: vectorization support: reference tmp has
                  aligned access   [ SIMD.c(605,17) ]
   remark #15389: vectorization support: reference db has
                  unaligned access   [ SIMD.c(605,17) ]
   remark #15381: vectorization support: unaligned access used
                  inside loop body
   remark #15427: loop was completely unrolled
   remark #15415: vectorization support: gather was generated
                  for the variable db:  strided by 1
                  [ SIMD.c(605,58) ]
   remark #15415: vectorization support: gather was generated
                  for the variable score_matrix:
                  indirect access [ SIMD.c(605,31) ]
   remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15458: masked indexed (or gather) loads: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 20
   remark #15477: vector loop cost: 2.180
   remark #15478: estimated potential speedup: 8.420
   remark #15479: lightweight vector operations: 8
   remark #15487: type converts: 2
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

Figure 5.11: Part of vec-report without `#pragma vector aligned`

Figure 5.11 shows a report from the code before adding the `#pragma vector aligned`. `Remark #15389` tells us that the memory pointed to by `db` might be unaligned. Even though `db` is allocated using `_mm_malloc` which is a *memory aligned* `malloc`, the pointer given to the function does not carry this information with it. The programmer have to help tell the compiler this by explicitly using `pragma`'s. By adding `#pragma vector aligned` before the for loop the compiler knows the data is aligned and unnecessary checks are skipped.

```
LOOP BEGIN at SIMD.c(604,11)
   remark #15388: vectorization support: reference tmp has
                  aligned access   [ SIMD.c(605,15) ]
   remark #15388: vectorization support: reference db has
                  aligned access   [ SIMD.c(605,15) ]
   remark #15427: loop was completely unrolled
   remark #15415: vectorization support: gather was generated
                  for the variable score_matrix:
                  indirect access    [ SIMD.c(605,29) ]
   remark #15300: LOOP WAS VECTORIZED
   remark #15458: masked indexed (or gather) loads: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 20
   remark #15477: vector loop cost: 1.870
   remark #15478: estimated potential speedup: 10.320
   remark #15479: lightweight vector operations: 8
   remark #15487: type converts: 2
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

Figure 5.12: Part of vec-report with `#pragma vector aligned`

The report presented in Figure 5.12 shows the report after adding the `pragma`. Now the remarks for the pointer `db` says that it has aligned access. With this addition the runtime significantly improves due to no unnecessary alignment checks, see Figure 5.13.

5.2.1 Auto Vectorization

The Xeon Phi possesses a variety of build-in features for auto vectorization. They are automatically enabled with the -O2 or higher compiler flag. A quick test with vectorization disabled shows that with a query of length 1115 and a range of 10 the GCUPS calculated is only 0.669! On average the application calculates 43.4 GCUPS for this query length when vectorization is enabled.

In some cases the compiler flag for optimization is enough to exploit the resources available, but especially on the Xeon Phi the programmer has to

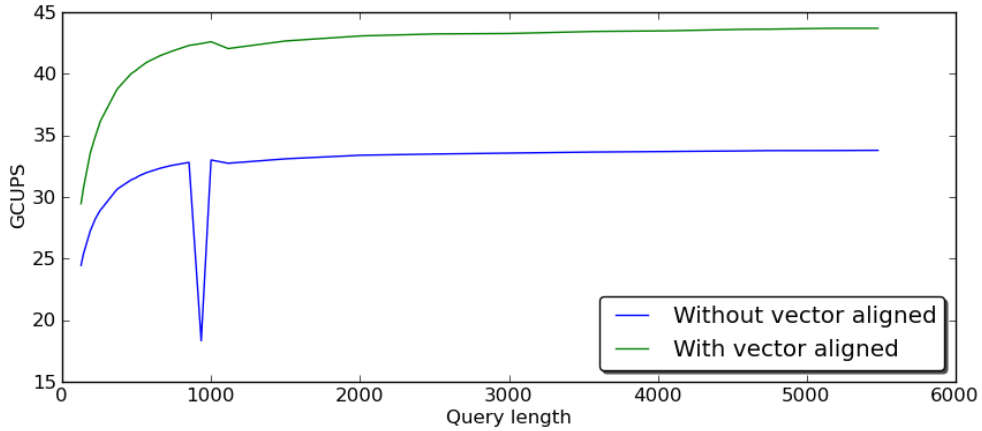

Figure 5.13: Auto vectorization using `#pragma`

help the compiler by stating where vectorization may be better utilized. To find this sections the vec-report compiler flag may be helpful.

Optimization with Compiler Flags

In some cases using the -O3 optimization flag may over optimize the code and yield a slower performance than with -O2. To test if this is the case for SWIMIC Figure 5.14 shows an execution with both.

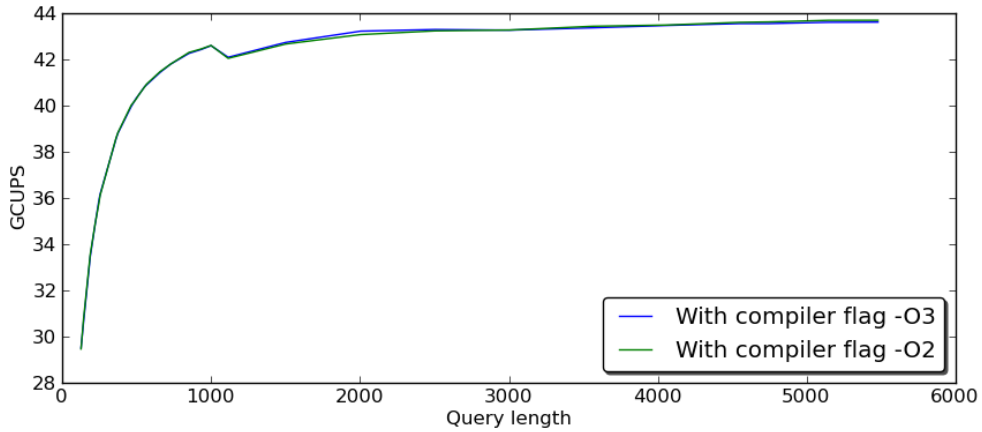

Figure 5.14: Optimization flag

As you can see, there is hardly any difference.

5.2.2 Memory Usage

To examine memory usage a few analysis of performance with different amount of calculations were carried out. The same tests also determine wherever the application is limited by either memory or compute power.

The first test examines the performance with twice the amount of computations as the regular algorithm. This helps determine if there are limitations in memory usage. If by doubling the calculations the performance is halved the application is only limited by computations and memory is no obstacle. On the other hand, if the performance shows no or small affection, the application is not limited by computation, but is bound by memory. From figure 5.15, that shows the performance analysis of twice the calculations one can see that the performance is influenced by the change in workload but not halved. This indicates that the application is to some extend bound by both memory and computation, but that there is still some potential performance gain not exploited yet in both areas.

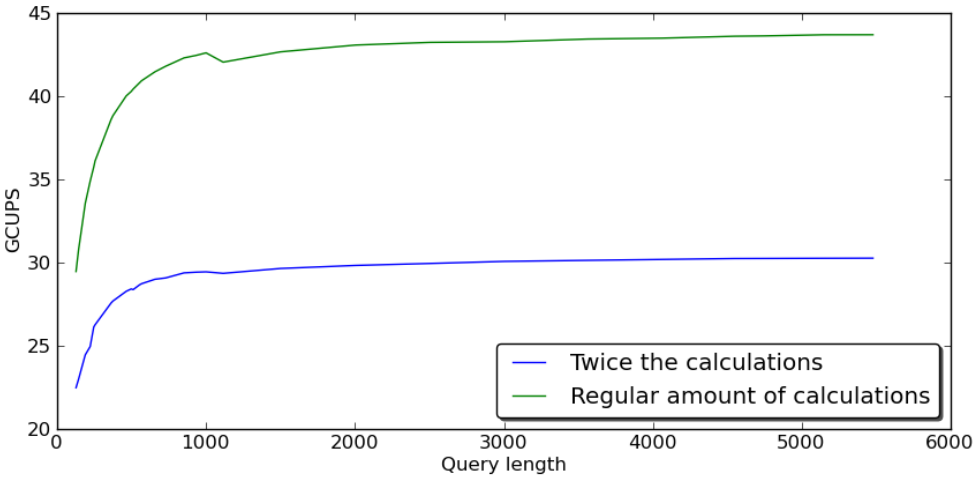

Figure 5.15: Performance analysis of doubling the calculations

For validity a counter test with no calculations was performed. In this analysis all memory is set up as used in the regular implementation, except the call to the `ALIGN` define (Figure 4.12 on page 34) is never called, hence no intrinsics are performed. The plot in Figure 5.16 showed a surprising result. How come the performance breaks down at a query length at approximately 800 amino acids? This behavior was not expected. However the analysis is highly valuable as it highlights a memory limitation. The only memory used in the Smith-Waterman algorithm [2] that varies with the length of the query is the `HE_array` used to store the previous column of the H and E array. This reveals that with long query sequences the saving of H and E values causes

cache exceedance and performance loss due to cache read misses.

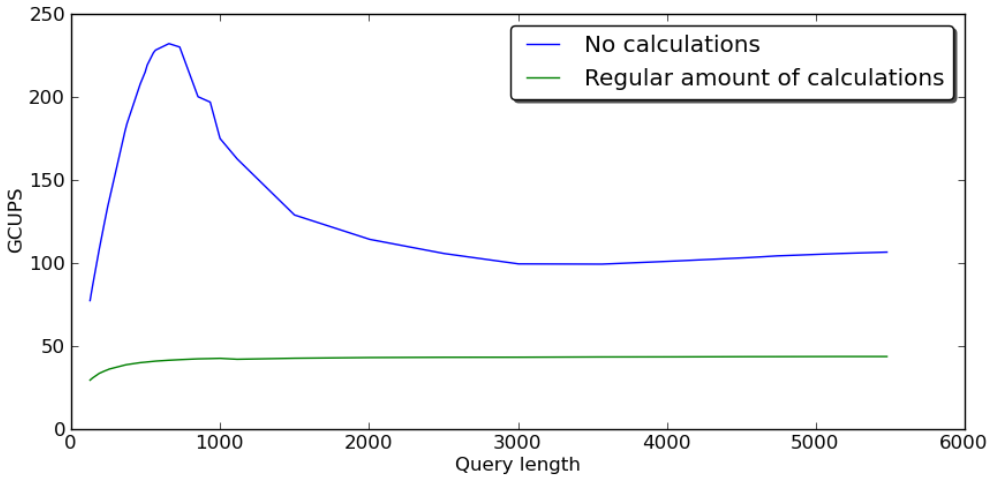

Figure 5.16: Performance analysis with no calculations performed

To verify this assumption a new memory analysis where executed. This time the saving of H and E values where done to the same fixed location in memory to examine the performance without a variable depending on the query length. The result is shown in Figure 5.17. As expected the performance continues to increase past a query length of 800 amino acids, thus the `HE_array` is a memory limiting factor.

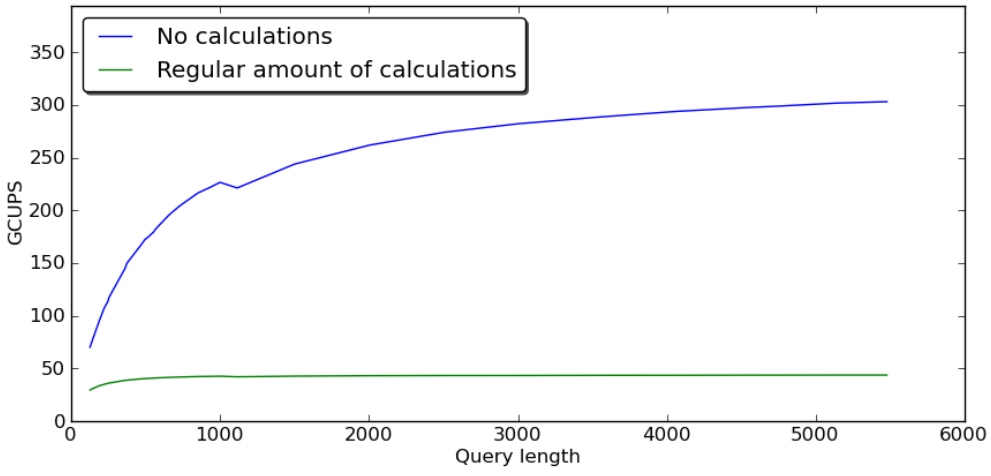

Figure 5.17: Performance analysis with no calculations performed and saving the result at the same location in memory

From a theoretical analysis calculating the cache usage the same result appear. The `HE_array` is 2 times the query length times the size of `int` for each database sequence.

$$HE\_array = 2 * q\_len * sizeof(int). \tag{5.1}$$

Since 16 sequences are aligned simultaneously in one vector the `HE_array` is multiplied by 16. There are T number of thread executing the calculations concurrently giving the total memory usage for the `HE_array`

$$MEM_{HE\_array} = HE\_array * 16 * T. \tag{5.2}$$

If the query length is 900 amino acids and the number of threads is 240 this gives

$$\begin{aligned} MEM_{HE\_array} &= HE\_array * 16 * T \\ &= 27648000 \\ &\approx 27MB. \end{aligned} \tag{5.3}$$

The local vectors used to store in between values such as F and SM is also needed in cache as well as the additional vector to save the calculated score. This is close to the 31 MB available cache total on the Xeon Phi coprocessor, and the result spotted in Figure 5.16 is proved.

To enhance the application to prevent this limitation from being a performance drawback a different approach on calculation order is necessary. Some alternative approaches including a box calculation is discussed in the future work chapter.

Utilization of Compute Power

As Figure 5.15 revealed, the application is limited to some extend by computation. This may be a result of order dependent calculations in the main loop in the Smith-Waterman algorithm sketched in Figure 4.12. The Xeon Phi are able to start executing a new instruction in the next clock cycle if the following instruction does not depend on the previous one. Figure 5.18 shows the optimal process of execution where the new execution starts in the next cycle.

Unfortunately in some cases the next calculation depend on the result of previous calculations and the processing unit is waiting to continue and computer power is gone to waist. In the worst case Figure 5.19 shows that the pipeline have to wait until the data from the first calculation is written back to memory before the execution is able to start. For SWIMIC this may be some of the reason for the compute limitation.

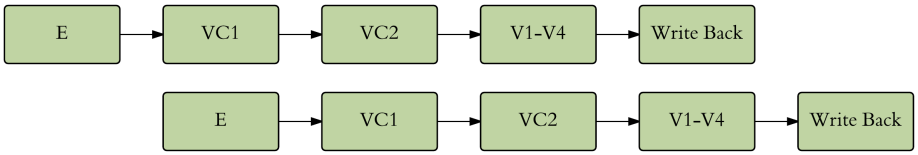

Figure 5.18: Optimal use of pipeline

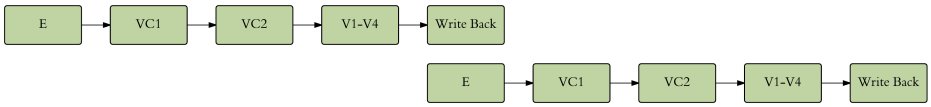

Figure 5.19: Dependant operations resulting in inefficient pipelining

5.3 Validity

To ensure the validity of the application the important aspects to look into is the calculated score of similarity. Due to the accuracy in the Smith-Waterman algorithm testing the validity of the calculated score is neglected. If the calculated total score matches the score calculated from for instance SWIPE [4], the algorithm is implemented correctly and further validation testing is not a necessity.

5.4 Comparison to Other Tools

Unfortunately due to limited time and resources a full examination against the competing tools was not possible. A good substitute is the comparison work done by Liu and Smith [3], comparing SWAPHI against SWIPE [4] and the BLAST+ [13]. These are not the initially intended tools to compare SWIMIC against, however they includes the two most interesting tools: SWIPE and SWIPHI.

The article [3] states: *" By using InterSP, SWAPHI achieves a performance of up to 58.8 GCUPS on a single Xeon Phi and up to 228.4 GCUPS on four Xeon Phis.*

Subsequently, we have compared SWAPHI to SWIPE (v2.0.7) and BLAST+ (v2.2.28) (see Fig 5.20 (b)). SWAPHI used the InterSP variant and each algorithm used its default scoring scheme. Additionally, SWIPE and BLAST+ used other options "-b 0 -v 0" and "-num alignments 0", respectively. On four Xeon Phis, SWAPHI could not outper- form BLAST+ on 16 CPU cores, but is superior to SWIPE on 16 cores. Compared to BLAST+ on 8 cores, SWAPHI performs better for most queries and runs 1.19 × faster on average (1.86 × maximally). Compared to SWIPE on 8 and 16 cores, SWAPHI gives a speedup of 2.49 and 1.34 on average

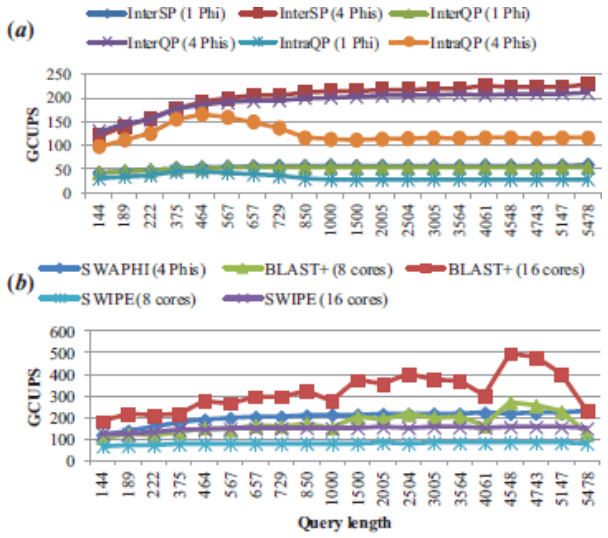

Figure 5.20: The comparison of tools done for SWAPHI

(2.83 and 1.52 maximally), respectively. "

Comparing SWIMIC to these results is a little disappointing. Despite a lot of effort, the performance accomplished does not measure up to the high level of performance accomplished by other tools. However SWIMIC performance compared to the performance of using a single Xeon Phi with SWAPHI is not as bad as it may look. SWIMIC's 43 GCUPS compared to the 58 GCUPS accomplished with SWAPHI is 74 %.

Tool	GCUPS achieved
CUDASW++ 3.0	119
SWIPE (dual 6 cores CPU)	106
SWAPHI (One Xeon Phi)	58
SWIMIC (One Xeon Phi)	43

Table 5.8: Leading tools and their achieved GCUPS

Table 5.8 presents the GCUPS achieved for the tools SWIMIC is compared to. The results are taken from each tools' own article.

Chapter 6

Conclusion

At the beginning of this thesis some areas of research was developed to set up the outline of the project ahead. Some of the areas came naturally while other faced unforeseen challenges that obstructed progress in parts of the development.

Implement and optimize a sequence alignment tool for the Intel Xeon Phi coprocessor

In this study a tool for local sequence alignment, SWIMIC, was implemented to run on Intel's Many Integrated Core Architecture (MIC) using the Xeon Phi coprocessor. The tool runs natively on the coprocessor and is implemented to utilize both the unique 512 bit vector unit and the shared memory between the four hardware threads per core.

Techniques used to optimize the application are on both thread and data level. For threading, the OpenMP API was used to distribute the workload among the 60 cores. The distribution is done dynamically during run time and the data being distributed is sorted to prioritize larger tasks. On data level, vectorization is used to utilize the compute power. Both SIMD intrinsics and pragma directives for auto-vectorization were utilized. Additionally, the database is preprocessed prior to the alignment process to attain a deliberate and optimal cache exploitation.

Examine whether the Xeon Phi coprocessor is a suitable hardware for sequence alignment

The study divided the research into two parts: the present days hardware, and the architecture itself.

Since the Xeon Phi used in this thesis, the Knights Corner, only supports vector operations on 32 and 64 bit, there is a lot of unused potential compute power due to the relatively small space needed to calculate each cell in the alignment matrix H. Other tools like SWIPE [4] uses only 8 bit operations to

calculate the alignment. Fortunately on the next generation of Xeon Phi, the Knights Landing, launching some time in the near future, 8 bit operations are supported, and this will make the Xeon Phi more suitable for sequence alignment.

When it comes to the architecture itself, the relatively high memory usage required to perform sequence alignments limits performance on the Xeon Phi, due to small local cache when all threads have independent tasks. This drawback will most likely always be present on future generations of the Xeon Phi, thus making the Xeon Phi a less suitable hardware to perform sequence alignment on.

Even with the 8 bit support, the memory usage will always limit the performance. The Knights Landing will be better, but never as good as optimized implementations for GPUs and CPUs. With more Xeon Phis working together the performance can measure up to GPUs and CPUs, however a consideration of accessibility and hardware expense is required.

Determine how well the implemented tool utilizes the unique coprocessor architecture

There are many important considerations when assessing SWIMIC's architecture utilization. Both memory utilization and how compute power are distributed is essential to look at. Due to the extensive use of memory in the Smith-Waterman [2], especially with Gotoh's [15] modification for affine gap penalty, the focus in this study has been memory prior to computation.

Despite considerable effort on memory exploitation, the results presented in Section 5.2.2 reveals that there is still unused potential in memory utilization. With the proper tools, such as VTune, this potential could be further exploited. Regardless of the absent profiler, some ideas for future work in this area is discussed in the next chapter.

When looking at computation, the performance of the application can still be improved, but only to a certain extent. Pipelining is something that could be explored further to utilize more of the potential compute power.

Determine whether or not the finish tool is competitive compared to other sequence alignment tools

It was quite obvious from early on that SWIMIC could not compete with existing tools design for other hardware alone. The only way a Xeon Phi application would be competitive with leading tools like SWIPE [4] and CUDASW++ [5] is by using multiple coprocessors. Preferably with some of the workload calculated on the host CPU.

On of the goals when starting this thesis was to reach, or even exceed the performance of the only published tool implemented for the Xeon Phi coprocessor, SWAPHI [3]. Unfortunately this did not happen. SWIMIC reached 74

% of SWAPHI's performance. SWAPHI uses an offload execution model that allows for distribution of some of the workload to the host CPU. Other advantages SWAPHI has over SWIMIC is the fact that the programmers implementing the application have more experiences with sequence alignment and knows a lot of optimization tricks used for other hardware. Both Yongchao Liu and Bertil Schmidt contributes on CUDASW in addition to SWAPHI.

Even with some potential memory and compute power still unexploited the performance relative to existing tools are disappointing. SWIMIC achieves 43 GCUPS at best, which is 74 % of a similar tool also running on the Xeon Phi, and only 40 % of the leading tool running on CPU's.

The study shows that the Xeon Phi coprocessor is not a suitable architecture to perform sequence alignments utilizing the Smith-Waterman algorithm on, due to relatively high memory footprint. The shared memory architecture possess a relatively small combined cache and with the lack of support for smaller data types there is a limitation that the four hardware thread and a 512 bit vector unit per core can not overcome.

Chapter 7

Future Work

7.1 Hardware

The first setback perceived was the lack of support on the Xeon Phi for small data types. With only vector support for `int` and `long`, the anticipated performance gain from doing four times as may calculations simultaneously than SWIPE's [4] 16 was shattered. Instead of calculating 64 8 bit alignments simultaneously in the 512 bit long vector unit, one had to accept defeat and perform only 16 32 bit operations. This lead to a major drawback since the new 512 bit vector unit was the reason for choosing the Xeon Phi as architecture for this thesis.

Fortunately in the near future Intel releases a new generation of the Xeon Phi coprocessor with the support of 8 bit intrinsics. In theory this implies a four time performance speedup from the implementation as it is today. Due to possible overhead issues with the 8 bit alignment a performance loss from the theoretical speedup is inevitable.

Probably a better solution is to use `short`'s of 16 bits, which will double the performance without the need for excessive overflow checks.

7.2 Improvements of the Tool

The implementation of SWIMIC is not perfect. There are still areas that may benefit from more tuning and further analysis, with for instance VTune, that can locate hotspots and unused resources. Other approaches of calculation order is also interesting to look further into due to limitation found in Section 5.2.2 concerning cache utilization.

7.2.1 Auto Vectorization

There are still a lot of unexploited pragma directives that might be suitable for SWIMIC. The performance on the Xeon Phi is relying on the compiler to

generate rapid and optimized code. Intel have encouraged programmers to make full use of the already optimized directives instead of implementing custom functionality. However, these advice were discovered at a late stage of this thesis work and a lot of the directives had to be left at a theoretical stage, rater than put into action. The pragma directives exploit in this project are the `#pragma vector aligned`, `#pragma parallel` and `#pragma omp for`.

Further exploitation with the `#pragma vector` may prove beneficial. The `vector` pragma has a lot of available parameters, and some of the others than the exploited `aligned` may prove a better performance. The `#pragma simd` also offers exciting options for vectorization and may be utilized on sections where the `vector` pragma fall short.

Efficient vectorization involves making full use of the vector hardware. This implies that the programmer should strive to get most code to be executed in the kernel-vector loop as opposed to peel loop and/or remainder loop.

A reminder loop is created to execute the remaining iterations when the number of loop iterations for a vector loop is not a multiple of the vector length. The reminder loop may be vectorized, but it wont be as efficient as the kernel loop due to masks, gathers and scatters instead of unit-stride or loads and stores. From the `vec-report` in Figure 5.12 on page 52 one can see that `remark #15458: masked indexed (or gather) loads: 1` is preset, and may be prevented with utilization of the `lopp_count` pragma that prepares the compiler of the remaining iterations.

A peel loop is generated by the compiler typically to align one of the memory accesses inside the loop. The peel loop never exceeds the length of a vector, but the peel loop itself, even though it may be vectorized, is less efficient. With correct use of the `#pragma vector aligned`, already used in SWIMIC, the peel loop should in theory not be generated. The `lopp_count` pragma may also influence the compilers decision of whether or not to create a peel loop.

Another option available to make better use of the vector hardware is to use safe padding. This is enabled by the compiler flag `-opt-assume-safe-padding` and determines whether the compiler assumes that variables and dynamically allocated memory are padded past the end of an object. This means that code can access up to 64 bytes beyond what is specified in the application. To satisfy this assumption, one must increase the size of objects when using this option. For instance, if memory is allocated by `malloc` an additional padding of 64 needs to be added manually. With this padding the compiler can more freely vectorize, resulting in a possible performance gain.

A higher performance may also be accomplished by making use of the `prefetching` directive. With prefetching of data to either L1 or L2 cache a

better cache utilization may be achieved. This is something the profiler might have helped detecting.

7.2.2 Inspection and Analysis

VTune

It was tedious not to get the profiling tool from Intel, VTune, to work. With this tool a lot of time used to examining whether a solution is utilizing the available resources may have been saved. To further improve SWIMIC some type of profiling seems indispensable. VTune looks promising on paper, thus making it the obvious tool to put more effort into getting to work for further implementation on SWIMIC.

Guided Auto-Parallelization Report

A compiler feature detected to late in this thesis work to look further into is the *Guided Auto-Parallelization* report. It is designed to report suggestions for loop optimizations and may give some useful tips on how to continue optimizing. The *Guided Auto-Parallelization* report is enabled with the compiler flag `-guide`, and may be extended by adding `-parallel` option as well. It is advantageously to use the *Guided Auto-Parallelization* report in combination with the `-vec-report` option described in Section 5.2 on page 51.

Assembly code inspection

Visual inspection of assembly code can help identify performance problems that may merit further investigation. This especially applies to for loops and how they are performed. From the assembly code one can for instance attain information of whether or not peel and/or remainder loops are created and usage of scatters or gathers. In addition the assembly code may indicate lack of prefetch instructions.

To obtain the assembly code for an application one can use the `-S` option when compiling. This will generate a file with the `.s` extension instead of the regular executable. The debugging flag `-g` is recommended discarded as it will remove a great deal of extra symbolic labeling in the assembly file and make the assembly code more difficult to read.

VTune possesses the ability to view the assembly code. If symbolic information is present a side-by-side view of the assembly code and the source code can be displayed. This will make the navigation of the rather arcane assembly code to a breeze compared to familiarize with code completely on your own.

7.2.3 Transition to Knights Landing

SWIMIC is implemented to tackle the transition to the new generation of Xeon Phi fairly well. The big known change that SWIMIC wants to take advantage of is the 8 bit support. In theory only a new define section at the top of the file `align.c` is needed. In this section the definition of the new intrinsics have to be done, in addition to define the `SW_MULTIPLE` to 64 instead of 16. This solution is how it is possible to run SWIMIC on a regular CPU with only 128 bit vector support.

Query Lengths

An improvement to balance out the performance difference between long and short query lengths are a natural issue to continue working on. Due to an average length of 300 to 400 amino acids per sequence in the database, it is disadvantageous to use a tool that does not reach its performance high until a query length of a 1000 amino acids. This seems to be a common challenge for most sequence alignment tools, but nonetheless a task to look into.

7.2.4 Different Approaches of Calculation

In the study of this thesis a couple of different approaches of calculation order was explored. Calculating additional database sequences proved to be poorer (Section 5.1.3), while executing a larger range of columns exceeded expected performance gain (Section 5.1.4). As revealed in Section 5.2.2, performance is limited by memory when calculating an entire column of H when the query length exceeds appropriate 800 amino acid entries.

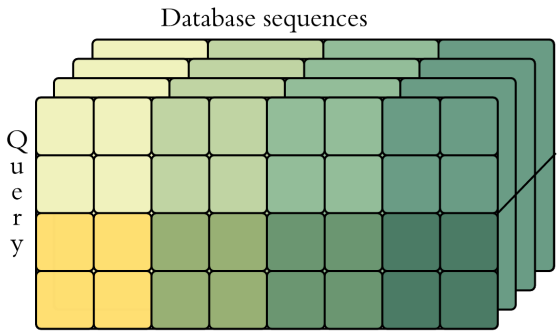

Figure 7.1: Calculation of a square

An approach of calculating a square of cells in the H matrix, illustrated in Figure 7.1, will solve this problem. Additionally the advantages concerning cache utilization from the multiple columns still applies.

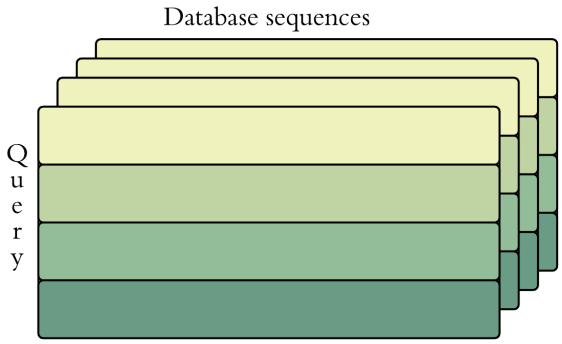

Figure 7.2: Calculating an entire row per iterations

Also interesting to examine is how performing row-wise calculations compare to column-wise performs. An illustration is presented in Figure 7.2. Both one row, and a larger range of rows, can prove efficient since data resides row-wise in memory.

Bibliography

1. Altschul, S. F., Gish, W., Miller, W., Myers, E. W. & Lipman, D. J. Basic local alignment search tool. *Journal of Molecular Biology* **215,** 403–410.
2. Smith, T. F. & Waterman, M. S. Identification of common molecular subsequences. *Journal of Molecular Biology* **147,** 195–197.
3. Liu, Y. & Schmidt, B. *SWAPHI: Smith-waterman protein database search on Xeon Phi coprocessors* in *2014 IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (), 184–185.
4. Rognes, T. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics* **12,** 221.
5. Liu, Y., Wirawan, A. & Schmidt, B. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics* **14,** 117.
6. Eidhammer, I., Jonassen, I. & Taylor, W. R. *Protein bioinformatics : an algorithmic approach to sequence and structure analysis* (Wiley, Chichester [etc.]).
7. Consortium, T. U. Update on activities at the Universal Protein Resource (UniProt) in 2013. *Nucleic Acids Research* **41,** D43–D47.
8. Benson, D. A., Cavanaugh, M., Clark, K., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J. & Sayers, E. W. GenBank. *Nucleic Acids Research* **41,** D36–D42.
9. Pruitt, K. D., Tatusova, T. & Maglott, D. R. NCBI reference sequences (RefSeq): a curated non-redundant sequence database of genomes, transcripts and proteins. *Nucleic Acids Research* **35,** D61–D65.
10. Lipman, D. J. & Pearson, W. R. Rapid and sensitive protein similarity searches. *Science* **227,** 1435–1441.
11. Barth, M., Sweden, K., Byckling, M., Finland, C., Ilieva, N., Bulgaria, N., Saarinen, S., Schliephake, M., Weinberg, V. & Germany, L. *Best Practice Guide Intel Xeon Phi* v1.1. Intel ().

12. Jeffers, J. & Reinders, J. *Intel Xeon Phi Coprocessor High-Performance Programming* 430 pp. (Newnes).

13. Camacho, C., Coulouris, G., Avagyan, V., Ma, N., Papadopoulos, J., Bealer, K. & Madden, T. L. BLAST+: architecture and applications. *BMC Bioinformatics* **10,** 421.

14. Needleman, S. B. & Wunsch, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* **48,** 443–453.

15. Gotoh, O. An improved algorithm for matching biological sequences. *Journal of Molecular Biology* **162,** 705–708.

16. Amdahl, G. M. *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities* in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (ACM), 483–485.

Appendices

Appendix A

The preparation Process

Even though access to the Abel Cluster's Xeon Phis were granted in the last stages of this thesis, initially only a Xeon Phi provided by SINTEF were to be used. A plethora of difficulitys arose while installing both hardware and software, and all these aspects are described in this appendix with all the frustration that arose as hidden as possible. In some ways this preparation process felt at times like the whole thesis and that the actual application created was just a simple little side mission, instead of how it was supposed to be, the other way round.

The only part of the setting up process that were done before this thesis work started was buying the necessary hardware and assemble all the components into the machine. Firstly it was tested with a different processor than the Intel Xeon, but the coprocessor would not even communicate with other processor, thus led to the conclusion that it was best to do exactly as Intel proposed regarding host processor, cooling system and cabinet.

A.1 Intel Software

To communicate with the coprocessor Intel's Manycore Platform Software Stack (MPSS) is required. For Ubuntu this not provided and some tweaking was necessary to get it properly set up. For compilation Intel provided a Parallel Studio XE 2015 student license to compile the code for a Many Integrated Core (MIC) architecture.

A.2 Library and Permissions

Some supplementary libraries both for OpenMP (Open Multi-Processing) and MPI (Message Passing Interface) were required to exploit the parallelism the Xeon Phi possesses. This libraries had to be copied to the coprocessor

from the host to be utilized. A simple task causing lots more trouble than anticipated. First there were no root access on the coprocessor, hence no permissions to copy the library files from the CPU into the shared library folder at root on the Xeon Phi. Setting PATH variables did not help, apparently only the /lib64 folder in root was looked at during runtime. The solution became to mirror all passwords and users from the host CPU over to the MIC operating system on the Xeon Phi and then use the root password created on the host computer to access the root on the Xeon Phi. This then made it possible to copy the needed files and finally things started to look brighter. Additionally the Intel parallel studio student license provided by Intel was not sufficient since it did not support MPI, therefore a new inquiry to Intel was needed to get a more extensive license.

With everything up and running the playing around with the Xeon Phi could finally begin. First some simple non-vectorized programs only adding numbers in for-loops, continuing to more advanced programs including an OpenMP parallelization to test the speed when using multiple threads per kernel, and then at last the prototype.

A.3 Debugging

A good debugger is usually a must when programming in C and the familiar error message `Segmentation fault` appear. Something is wrong but where the fault is, is a mystery. A debugger is used to locate "traps" in the code where the program cannot normally continue because of a programming bug or invalid data. For example, the program might have tried to use an instruction not available on the current version of the CPU or attempted to access unavailable or protected memory. A debugger might also be used to analyze values and steps of a program from a given point, e.g. a from a given breakpoint. When the program "traps" or reaches a preset condition, the debugger typically shows the location in the original code if it is a source-level debugger or symbolic debugger, commonly now seen in integrated development environments.

gdb is a handy tool to troubleshoot when problems occurs in C, but is this tool available on the Xeon Phi? Intel as always promise on there page that things are easy, and the use of *gdb* should not be a problem. After several attempts and a lot of different guides online, *gdb* was left abandoned and the problem got solved with good old thinking and going through the implemented logic yet a handful of times.

A.4 Profiling

An important step in optimizing an application is to perform a profiling, which is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. This to locate bottlenecks in the source code, which is a good indicator on where to optimize next. The analysis is achieved by instrumenting either the program source code or its binary executable form. Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, and performance counters.

For the Xeon Phi, Intel provides a powerful tool, VTune, to make a thorough analysis of all aspects of running an application on the coprocessor. It has a easy to use and explicit user interface that displays all the gathered data.

Unfortunately like all the other Intel tools that where installed, it did not workout as easily as Intel said it would. Who would buy a product if they warned the user beforehand that it was not running smoothly? When trying to do a profiling there appeared a list of warnings and the displayed result had no connection with the desired application. Since the application being analyzed was an implementation running natively on the coprocessor the profiler for some reason only analyzed the SSH login to the coprocessor process instead of the Xeon Phi application.

Even though access to the Abel Cluster's Xeon Phis where granted and executions performed more smoothly, a profiler was not preset, thus SWIMIC is never profiled.